LEARNING CONTROL OF PRODUCTION SYSTEMS

Salvatore J. Stolfo

Columbia University

CUCS-6-79

ABSTRACT

One of the central problems in Artificial Intelligence is that of designing appropriate and efficient mechanisms for representing and learning real-world knowledge. We propose a scheme composed of a declarative form, <u>production systems</u>, controlled by a separate collection of procedural information, a <u>control language</u>, together with a debugging facility, <u>meta-rules</u>. We show that it is feasible to infer the control information from an analysis of traces of the successful executions of the production system program provided by a human trainer, thereby creating a system capable of improving its performance by experience. We outline the representation scheme and the inference algorithms, and demonstrate the approach with an example.

INTRODUCTION

During the last few years, a number of relatively effective Artificial Intelligence (AI) programs have been written incorporating huge amounts of problem-specific knowledge acquired directly from human experts (for example [5]). Consequently, a significant portion of the design of such systems must be devoted to mechanisms which facilitate this knowledge transfer. The search for effective knowledge representations and acquisition procedures have become central problems for AI.

By now it has become common to distinguish between declarative information, which can be thought of as knowing what, and procedural information which can be thought of as knowing how. Declarative representations offer the advantage of being easier to acquire: the decomposability of such information is readily modifiable and extendable [6,16,24]. Unfortunately, a straightforward implementation of such information corresponds to a nondeterministic program which makes a relatively blind search through the solution space. Procedural representations offer the advantage of being efficiently executed by a machine. However, procedures are hard to debug and modify and therefore are not as easy to acquire.

Not surprisingly, many researchers have focussed attention towards schemes incorporating procedural or control information (plans, strategies, goals) into a primarily declarative framework. By fine tuning the procedural components to suit the needs of the problem domain, effective performance has been achieved in several cases. This contrasts with the earlier approaches in AI were a few general control regimes were applied uniformly to a declarative form and which met with very limited success. Some of the earliest examples of these dual representation frameworks include partial evaluation of predicates in predicate calculus theorem proving [1] and the geometry filter of Gelernter [9]. More recent examples include Clause Interconnectivity Graphs [18] in a resolution theorem-proving framework, Robot Plans in the STRIPS problem-solving system [7] and procedural attachment in frame representation systems [19]. Many schemes have been proposed for embedding control information into Production Systems [6,14], for example, annotated production systems [11], Petri-nets [26], and semantic nets [15]. In [16], Rychener discusses the approach of building a rational goal structure into a production system while Davis [5] favors a separate uniform set of meta-rules specifying local control information in a hierarchical fashion.[1]

In this paper we will focus on the problem of representing and acquiring procedural knowledge from a human expert. We have developed a representation scheme composed of a production system formalism, encoding the problem-specific information in declarative form, together with a control language specifying permissible sequences of rule applications, and a set of meta-rules used as a debugging aid for the control language. In [10], Georgeff independently proposes a similar scheme and discusses many of its implications. (See [17] also.) We give the details of this representation in later sections of this paper.

Separating control completely from the declarative form (productions are disallowed from containing tags or 'control elements' to signal other productions) allows us to focus on mechanisms for manipulating, and in particular learning this information. This also allows for the possibility of using the same set of declarative statements in different ways for a variety of problem tasks by varying the procedures used.

[1]The idea of using a uniform multi-layered production system is not new, although the specified use of the meta-rules in Davis' system is different. The Algol-68 specification uses a grammatical form where one set of context-free rules specify a possibly infinite language of non-terminal symbols. Members of this language are substituted in a set of lower level template rules, also in context-free form. This creates a possibly infinite set of rules specifying the Algol-68 language. This scheme is equivalent to general phrase-structured grammars and is capable of specifying the context-sensitivity of Algol-68. Davis' meta-rules are used for control only, ordering the set of lower level rules during the selection process, conflict resolution, and do not alter the form or content of these rules.

In many of the schemes cited above, the control information is acquired by explicit specification by a human expert (learning by being told). This presupposes that human experts can readily describe their methods and procedures for solving problems as easily as say a medical doctor can enumerate the formal names of the bones in one's arm.

It is our belief that this type of information cannot be easily communicated this way. Instead, it seems preferable that <u>control information be effectively learned or inferred from observing the behavior of the human expert when solving example problems</u>. In one's own experiences, in fact, learning by observation or example is probably the dominant way information of this type is conveyed. (Learning how to program or prove theorems, and the phrase we all have heard as youngsters when our parents were angry with our behavior, <u>Don't do what I do</u>, <u>do what I say</u>!, exemplify this type of learning.)

Sickel's work [18] is an example of automatically inferring control information from a syntactic analysis of the declarative form. The work of Fikes, Hart and Nilsson is closer to what we mean by learning control from observation. In the STRIPS problem-solving system, sequences of operator applications used in the solution of simpler problems were stored (along with data flow information) for possible guidance (or planning) in solving more difficult problems. (The work we report stresses the importance of a deeper analysis of the sequence of operator applications.) However, the STRIPS system analyzes sequences produced by the problem-solving system alone when solving simpler problems. We have decided to infer this information from traces supplied by a human expert guiding the program in action. This is a simpler problem since the traces we analyze contain much more useful information than traces found by trial and error search, especially if the trainer is a good teacher. The work of Biermann [4] is also of this form.

It can be argued that with this scenario the declarative components while constructed by the human expert inevitably contain some control information that is thrown away and then rederived later with considerable effort. We claim that the control information will be much more accurate and clearly formulated when a specific example is presented and its solution is displayed and examined. Prespecifying control would invariably need modification. (Hand simulation or several runs on sample data remains the best way to uncover bugs in a computer program.) Furthermore, this might not allow for different uses of the same information for a variety of problems.

In general, we might expect that the control information will embody very sophisticated principles which are inferable only by the use of considerable intelligence. For example, it is clear that the deduction of the heuristic principles of evaluation and alpha-beta search from a declarative chess program which specifies only the rules of chess will require an analysis which is considerably beyond the ability of present techniques; even the optimization of the parameters in a linear evaluation function involves highly sophisticated processing.

However, it is the contention of this paper that there are a number of important areas in which it might be possible to deduce control information automatically. These include declaratively specified problems:

- for which there exists a relatively simple algorithmic procedure
- whose performance can be improved in frequently occurring or particularly important special cases
- in which particular subproblems can be solved by simple algorithmic procedures.

In the following sections we present our approach in detail along with a problem that our system was applied to with good success.

APPROACH

Our approach is as follows:

1. Select a 'typical' input to the program and run the program repeatedly.

2. Guide the selection of appropriate knowledge (rules) when a conflict arises.

3. Record the sequence of rules selected together with input/output information and the conflict set of rules on each cycle.

4. Describe the better (i.e. shorter) successful sequences in a control language, CL/1, designed for this purpose (and described in the next section).

5. Generate a set of meta-rules whose objective is to aid the CL/1 description if the sequencing is inappropriate.

6. Use the CL/1 description and the meta-rules to guide the program's subsequent decisions.

In view of the complexities of the technical problems with this approach, it was necessary to define a starting point from which we can proceed to study the more general case. Accordingly, we have made certain assumptions and decisions in order to derive useful results.

Inherent in our approach is the assumption that good decision-making procedures or heuristics can be inferred from the performance of the program on only selected inputs. We anticipated that the selection of inputs would be critical and that eventually new inputs would be handled incrementally, as was done by Winston [25]. However, in this work we concentrate on the simpler problem of getting a good solution for the nonincremental case. In general, as suggested by work on learning systems [25] and information theory [12,20], we give preference to short CL/1 descriptions which will generate a high proportion of short successful solution sequences and few long or unsuccessful sequences.

The human expert running the program in 'training mode' is aware of the internal structure of the program. In subsequent generalization of this approach, we anticipate the necessity of using techniques similar to those of Davis [5], which will enable the trainer to deal only with the external behavior of the program.

We also have deliberately chosen to exclude information about sequences which end in failure. It is clear that, as found by Winston and others [3,25], counterexamples will be extremely valuable. However, as the reader will note below, even the simpler problem we study poses considerable technical difficulties and it was our feeling that a clearer picture would emerge from the simpler approach. Furthermore, the experiments described below suggest that useful results can be obtained without counterexamples.

The description language, CL/1, is modelled after Regular Expressions. Therefore, the technical problem that we are faced with can be stated as the construction of the minimum length Regular Expression which agrees with the sample data (solution sequences). Unfortunately, this problem has been proven to be NP-complete by Angluin [2]. The approach we use is therefore heuristic in nature.

In the following sections we outline the details of the representation scheme and then present the inference algorithms which analyze the solution sequences.

THE PRODUCTION SYSTEM LANGUAGE

The declarative information acquired from the human expert is written in Production System (PS) form [6,8,14,16]. A PS is a (nondeterministic) program consisting of a set of _productions_ or _rules_, called _Production Memory_ together with a data base of assertions, called _Working Memory_ (WM). Each rule consists of a conjunction of patterns of data elements, called the _left-hand side_ (LHS) and a series of actions called the _right-hand side_ (RHS). The RHS specifies information that is to be deposited in or removed form WM. Execution of the program consists of repeating the following actions (each iteration is called a _cycle_):

1. for each rule, determine if its LHS matches the current environment in WM (multiple instantiations are possible).

2. from the set of rules satisfying step (1), called the _Conflict Set of Rules_, nondeterministically select one.

3. _fire_ the rule selected in step (2), that is, apply the actions specified in the RHS of the rule.

In the exact form of the PS representation we use, called PROSYS a variant of CPS2 [3], data elements can be any LISP data structure. An atomic data element in the LHS of a rule must match an exact data element in WM and a list must match a list with the same structure and content. A symbol preceded with an equals sign (=) represents a variable (existentially quantified) which can match any data structure. A symbol preceded by * can match any data structure not equal to the data structure matched by a corresponding variable prefixed by an equals sign. The & symbol has the same function as the SNOBOL4 immediate assignment operator. When a rule is fired, the matching data elements are not deleted from WM unless they are included as arguments to the <delete> system function in the RHS of the rule. The other system functions are represented in lower case and enclosed in pointed brackets (< >). Their function is described by their names. The operator - in the LHS has the same function as <not> (the associated data element is not contained in WM). Finally, the symbol ! is an operator which matches the entire remaining portion of the list that contains it. Where it appears in the RHS, it deposits the list matched in the LHS but without the enclosing parentheses.

The following set of productions is a portion of a general robot problem-solving program that our system was applied to. The entire program contains 36 productions. The robot is modelled as a) having a single eye with which to focus on a single object, b) a hand with which to grasp an object and c) a limited memory which can remember a single object, a single pile of objects and a particular color at any one time. The particular problem we focussed on is a jigsaw puzzle task. Some of the productions are general in nature and allow for the piling of physical objects and through the functions of the eye, hand and memory, the systematic scanning of an ordered pile of objects. Other productions are unique to the jigsaw puzzle domain. Various sensing productions are used to indicate a variety of conditions of WM, displayed to the trainer but which do not alter the contents of WM. In this representation, we ignore spacial considerations so that we may concentrate on the problem of learning procedures.

Production Memory

```
   LOOK-AT-OBJECT-ON-TABLE ·
       ((LOOKING-AT =ANYTHING) & =C1
        (ON-TABLE  =OBJECT)
       -(LOOKING-AT =OBJECT)
       -(ON-TOP-OF =ANY =OBJECT)
                             -->
                      ((<delete> =C1)
                      (<write> |I'm now looking at|  =OBJECT)
                      (LOOKING-AT =OBJECT))
   PICK-UP-OBJECT-IN-VIEW-TABLE
       (HANDEMPTY & =C1
       (LOOKING-AT =OBJECT)
       (ON-TABLE =OBJECT) & =C3
      -(ON-TOP-OF =ANY =OBJECT)
       (NUMBER-IN-HEAP =NUMB) & =C4
                             -->
                      ((<delete> =C1 =C3 =C4)
                      (NUMBER-IN-HEAP (<SUB-1> =NUMB))
                      (<write> |I just picked up | =OBJECT)
                      (HOLDING =OBJECT))
   PIECE-HAS-STRAIGHT-EDGE
       ((LOOKING-AT =P)
       (SIDE =ANY =P =SIGN  0. =ANYC)
                             -->
                 (<write> |Piece | =P | has a straight edge|))
   PIECE-FITS-IN-PUZZLE
      ((LOOKING-AT =OBJ)
       -(IN-PUZZLE =OBJ)
        (SIDE =ANY =OBJ =SIGN =N =ANYCOLOR)
        (IN-PUZZLE =OBJECT2)
        (SIDE =ANY2 =OBJECT2 *SIGN =N =ANYCOLOR2)
                             -->
                     (<write> |The | =ANY | side of piece |
                   =OBJ | fits the | =ANY2  | side of piece |
                   =OBJECT2 | which is in the puzzle|))
   PUZZLE-IS-FINISHED
        ((NUMBER-IN-PUZZLE =N)
        (NUMBER-OF-PIECES =N)
                             -->
                    (<write> | The puzzle is now complete!|)
                    (<halt>))
```

```
MAKE-A-PILE
    ((HOLDING =OBJECT) & =C1
     (LOOKING-AT =OBJECT)
     (NUMBER-OF-PILES =M) & =C3
     (ALL-PILES ! =R) & =C4
     NO-CURRENT-PILE & =C5
                          -->
                  ((<delete> =C1 =C3 =C4 =C5)
                   (NUMBER-OF-PILES  (<ADD-1> =M))
                   (ALL-PILES (<ADD-1> =M) ! =R)
         (<write> |I just created a new pile called pile |
                            (<ADD-1> =M))
                   (CURRENT-PILE  (<ADD-1> =M))
                   (IN-PILE  (<ADD-1> =M) =OBJECT)
                   (ON-TABLE =OBJECT)
                   HANDEMPTY)
PICK-A-PILE
   (NO-CURRENT-PILE  & =C1
    (ALL-PILES =P1 ! =R)
                        -->
              (<WRITE> |I am now working with pile | =P1)
              (<DELETE> =C1)
              (CURRENT-PILE =P1))
PUT-OBJECT-IN-PILE
   ((CURRENT-PILE =P1) & =C1
    (HOLDING   =OBJECT)  & =C2
    (LOOKING-AT =OBJECT)
    (ALL-PILES =P1 ! =R)
    (IN-PILE =P1 =OBJECT2)
   -(ON-TOP-OF =ANY =OBJECT2)
                        --> =C1
                        (<WRITE> |I just put | =OBJECT
                                 | on top of pile | =P1)
                        (<DELETE> =C2)
                        HANDEMPTY
                        (ON-TOP-OF =OBJECT =OBJECT2)
                        (IN-PILE =P1 =OBJECT))
FORGET-CURRENT-PILE
   ((CURRENT-PILE =P1) &=C1
                        -->
        (<DELETE> =C1)
        (<WRITE> |Pile | =P1 | is no longer being used|)
        NO-CURRENT-PILE)
```

```
CONSIDER-ANOTHER-PILE
   ((ALL-PILES =P1 =P2 ! =R) & =C1
    NO-CURRENT-PILE
                          -->
        (ALL-PILES =P2 ! =R =P1)
        (<DELETE> =C1)
        (<WRITE> |The next pile to consider is pile| =P2)
LOOK-AT-FIRST-IN-PILE
   ((LOOKING-AT =ANYTHING) & =C1
    (CURRENT-PILE =P1)
    (IN-PILE =P1 =OBJECT  & *ANYTHING)
   -(ON-TOP-OF =ANY =OBJECT)
                         -->
            (<DELETE> =C1)
            (<WRITE> |I am now looking at object | =OBJECT
                     | on top of pile | =P1)
            (LOOKING-AT =OBJECT))
REMEMBERED-OBJECT-IN-VIEW
    ((REMEMBERED-OBJECT =OBJECT)
     (LOOKING-AT =OBJECT)
                          -->
                         (<WRITE> =OBJECT | which is tagged|
                                  | as special is in view|))
```

THE CONTROL LANGUAGE CL/1

The CL/1 language provides a semantic framework with which to specify or describe sequences of rule applications in the execution of the PS program. The basic primitive of CL/1 is called a unit. A unit specifies either a rule application (with preconditions), in which case it is called a simple unit, or a control operation applied to a sequence of units. The control operations are: Permutation of a set of sequences, Alternative or conditional selection of a sequence from a set of sequences, Repetition of a sequence controlled by simple Boolean assertions (described below) in Disjunctive Normal Form (DNF) and (implicit) Concatenation of units producing sequences.

The control primitives are represented syntactically in Cambridge form by P.*, A.* and R.*, respectively, while concatenation is represented by a list of units enclosed in double pointed brackets (<< >>).

The CL/1 operators correspond to various control primitives of conventional programming languages, and to that of Regular Expressions (where permutation corresponds to a shuffle operator). The choice of using Regular Expressions for control depends on several considerations. First, people generally use descriptions of their own actions which appear very much like Regular Expressions. Secondly, since they are one of the simplest formalism, it would appear that they would be easier to induce from examples than other more complicated formalisms. Lastly, they are easily implemented and easy to understand.

However, it would appear that Regular Expressions are too limited in their expressive power to be of much interest. However, coupled with a powerful PS program as we use here, the total system is at least as powerful as the PS representation and is capable of a wide range of behavior with the additional control constraints. For example, consider the following example taken from Georgeff [10] (interpreting this production system in the usual formal grammar sense):

1. P1: S--> ABC
2. P2: A-->aA
3. P3: B-->bB
4. P4: C-->cC
5. P5: A-->a
6. P6: B-->b
7. P7: C-->c

Beginning with the initial sentential form (WM) containing S, these productions generate the language $\{a^i b^j c^k \mid i,j,k >= 1\}$ which is context-free. If we restrict the permissible sequence of rule applications to be a member of the language generated by the Regular Expression:

(1) P1 (P2 P3 P4)* P5 P6 P7 ,

then the language generated is $\{a^n b^n c^n \mid n >= 1\}$ which is context-sensitive. Notice that P2, P3 and P4 can be used in any order as can P5, P6 and P7. We can describe these additional control constraints in CL/1 as:

(2)<<P1 (R.* (P.* <<P2>><<P3>><<P4>>)) (P.* <<P5>><<P6>><<P7>>) >>.

Georgeff describes the use of Regular Expression control in this fashion to both limit the number of productions to be tested on each cycle and to leave nondeterministic selection points in tact and at well specified points within the control. For example, in (1) above, at the end of the repetition, only P2 and P3 can enter the conflict set of rules (severely limiting the number of productions to be tested, which can obviously be useful for large systems, but also too restrictive in general). This approach leaves the final decision as to which production to select up to the conflict resolution strategy (or meta-level knowledge base, see [5]).

There is some attempt in CL/1 to lessen the responsibility of the selection strategies built in to the PS interpreter by allowing explicit specification of conditions under which repetitions should be allowed and alternatives should be selected. The repetition operator in CL/1, therefore, contains both a While and Until clause (written in Cambridge form as W.* and U.*) and the alternation operator contains conditional expressions for each alternative very much like the LISP COND. Further, even the simple unit, which specifies the next rule to apply, contains a precondition for that rule to be selected. In total, this wealth of specified conditions is intended to move many of the nondeterministic decisions out of the PS interpreter and into the control mechanism *explicitly*.

But what might these conditions be? We could build a world model (representing meta-level knowledge) in addition to the machinery at hand, but this introduces additional complexity and ambiguity.

However, since the PS is defined by a human expert, we assume that the program is an accurate model of the problem domain (it can after all solve problems). Furthermore, the state of the PS program during execution is an accurate model of the state of the world viewed by the expert when solving problems. Therefore, the conditions we have defined are expressions testing the state of the PS program and in particular the contents of WM. Conjoining all of the data elements would clearly be useless, instead we would probably need to know what relationships exist between the elements. The problem then becomes that of choosing the correct relationships to describe, which is analogous to choosing the correct features of an object in pattern recognition (and indeed lies at the heart of the knowledge representation problem). (The TEIRISIAS-MYCIN approach is to interrogate the human expert to make explicit what the relationships might be in context.)

Consider the rule: P: (C1 C2 ... Cn --> A1 A2 ... Am). From the point of view of the human expert, this rule states that if the data elements C1, C2, ... Cn are contained in the data base at the same time, some special effect in the system should result. That is to say, there is a _strong relationship_ between those elements, which _represents_ an important feature of the state of the world, and the problem-solving system. (Of course, a different set of

productions could have a major impact on this approach, but we are relying on the expertise of the human trainer to decide what is important and what is not in the context of solving real problems.) Therefore, if we wish to describe the state of the system, describing the list of rule names whose LHS's match the data base appears to be useful. The conditions we use are exactly of this form but allowing for several list of rules in disjunctive form (written as OR.*). (The latest version of the language contains additional information on how the instantiations of the productions share matching data elements.)

The human expert is allowed to view only the conflict set of rules during training and is forced to make decisions based solely on this information. If this proves inadequate (which it has at times), dynamic additions to production memory are allowed at any point during execution. These changes become permanent additions to the knowledge base. A natural consequence of this approach is the use of _sensing_ productions (which output information without changing WM) and production names reflecting the contents or intended meaning of the rule as evidenced by the examples in this paper. By virtue of being embedded (ordered) in a sequence and since each sequence is an argument to a control operator, each condition has some implicit notion of a global context in which it applies. Examples of the CL/1 operators with conditions follow.

When executing the simple unit:

```
1. (PUZZLE-IS-FINISHED
        (OR.* (HEAP-EMPTY)(CURRENT-PILE-IS-EMPTY)))
   the production PUZZLE-IS-FINISHED would be selected if
   it is a member of the conflict set along with either
   of the productions HEAP-EMPTY or CURRENT-PILE-IS-EMPTY.

2. (R.* (W.* (PIECE-HAS-STRAIGHT-EDGE))
        (U.* (REMEMBERED-OBJECT-IN-VIEW))
      <<(LOOK-AT-NEXT-IN-PILE
                     (REMEMBER-CURRENT-OBJECT))>> )
   produces the sequence of simple units:
     <<(LOOK-AT-NEXT-IN-PILE (REMEMBER-CURRENT-OBJECT))
       (LOOK-AT-NEXT-IN-PILE (REMEMBER-CURRENT-OBJECT))
                .
                .
                .
                >>
   while the production PIECE-HAS-STRAIGHT-EDGE is active
   and REMEMBER-CURRENT-OBJECT is inactive.  The conditions
   are tested prior to producing the sequence on each
   iteration.  Notice that the conditions of the simple units
   contained in the argument sequence must still be satisfied
   when executing the argument sequence.

3. (P.*
    <<(LOOK-AT-FIRST-IN-PILE ())(LOOK-AT-NEXT-IN-PILE ())>>
    <<(FORGET-REMEMBERED-OBJECT ())>> )
   could produce three sequences:
           {<<(LOOK-AT-FIRST-IN-PILE())
               (LOOK-AT-NEXT-IN-PILE())
               (FORGET-REMEMBERED-OBJECT())>>
            <<(LOOK-AT-FIRST-IN-PILE())
               (FORGET-REMEMBERED-OBJECT())
               (LOOK-AT-NEXT-IN-PILE())>>
            <<(FORGET-REMEMBERED-OBJECT())
               (LOOK-AT-FIRST-IN-PILE())
               (LOOK-AT-NEXT-IN-PILE())>>  }
   In a parallel environment both sequences could be executed
   simultaneously. In  the actual implementation, the argument
   sequences are concatenated and  executed in order.

4. (A.* (PIECE-HAS-STRAIGHT-EDGE) <<(PICK-UP-OBJECT-IN-VIEW())>>
        (REMEMBERED-OBJECT-IN-VIEW) <<(CONSIDER-ANOTHER-PILE( ))  >>  )
   produces the sequence <<(PICK-UP-OBJECT-IN-VIEW())>> if
   PIECE-HAS-STRAIGHT-EDGE is active, otherwise
   <<(CONSIDER-ANOTHER-PILE())>> if REMEMBERED-OBJECT-IN-VIEW is
   active, or it produces no sequencing at all.
```

IMPLEMENTATION OF CL/1

The implementation of the operators is very straightforward. The PS interpreter maintains a stack of units corresponding to the CL/1 description that is in control of the sequencing. Successive units are popped from the stack and applied. If the stack is empty, or the precondition of a simple unit is false, the meta-rules are called. The algorithm which follows is invoked whenever a rule is to be selected from the conflict set. The details of the meta-rule implementation can be found in the next section.

META-RULES

There are four types of meta-rules which assist a CL/1 description in controlling a PS program. It is the simple unit which actually selects the next rule to fire on each cycle (the higher level control units produce sequences of simple units). If the DNF expression evaluates to false or the specified rule name is not active, the meta-rules are called upon to suggest a list of rules to try in order to force the DNF expression to evaluate to true. (We use the control in an irrevocable fashion, not wanting to resort to backtracking.) For example, suppose that the simple unit (A (B C)) is in control, E was the previously fired production, and the current conflict set of rules is {B D}. This situation may be described as

1. A and C should be active
2. D should (perhaps) be inactive
3. {B D} are currently active
4. E was just fired.

Accordingly, the meta-rules which we have developed are designed to deal with the four cases listed using the primitive functions Want-active, Want-inactive, Currently-active and Just-fired, respectively. In each case, a meta-rule may suggest a list of rules to try in that situation using the primitive function Try-to-fire. The suggestions are weighted since a rule may be suggested several times by different meta-rules. Each meta-rule is scanned and its LHS tested; if it evaluates to true, the corresponding RHS is

Figure 1: Implementation of CL/1

```
repeat forever; Apply the rule selected as follows:

   case;

      (Stack Empty):
        Set P to meta-rule choice;
        If P not empty then select P from conflict set;
        else stop;  end if;

      (Simple unit on stack):
        If the DNF of the unit is true and the rule name is active
           then
             pop the stack;
             select the rule name from the conflict set;
           else
             set P to the meta-rule choice;
             if P is not empty then select P from conflict set;
               else pop the stack; end if;
           end if;

      (Permutation unit on stack):
        pop the stack;
        push the appended argument sequences on stack;

      (Repetition unit on stack):
        if the While component, and the Until component are both
           empty then
           pop the stack;
           push argument sequence on stack; /* executed once */
          else
           if While condition is true and Until is false then
             push argument sequence on stack;
            else
             pop the stack;
           end if;
         end if;

      (Alternation unit on stack):
        Scan each argument of unit;
          if the DNF condition is true then
            pop the stack;
            push the corresponding argument sequence;
          end if;
        end scan;
        if none were true then pop the stack;
   esac;

end repeat;
```

executed. The exact definitions of the meta-rules follow. See figure 2 for examples of each type produced from the jigsaw puzzle system.

Let

1. P' be a list of rule names
2. J be the rule just fired on the previous cycle
3. D be the union of all the names appearing in the DNF of the current simple unit in control
4. U be the rule specified by the current simple unit
5. C be the current set of active rules.

The primitive function Want-active is a Boolean function which tests the current state of the CL/1 description and the currently active set of rules. It is defined as:

$$(\text{Want-active } P') = \begin{cases} \text{true if } P' \subseteq D \cup \{U\} \\ \text{else false} \end{cases}$$

Similarly, the other primitive functions are defined as:

$$(\text{Want-inactive } P') = \begin{cases} \text{true if } P' \subseteq C - (D \cup \{U\}) \\ \text{else false} \end{cases}$$

$$(\text{Currently-active } P') = \begin{cases} \text{true if } P' \subseteq C \\ \text{else false} \end{cases}$$

$$(\text{Just-fired } P') = \begin{cases} \text{true if } J \subseteq P' \\ \text{else false} \end{cases}$$

Figure 2: Sample Meta-rules

```
Type 1 Meta-rules:

 [(<WANT-ACTIVE>
              (THERE-ARE-NO-PILES)
                   ) -->
                       (<TRY-TO-FIRE> (DESTROY-A-PILE))]
 [(<WANT-ACTIVE>
              (FORGET-CURRENT-PILE
               PILE-IS-EMPTY
               DESTROY-A-PILE
               REMEMBER-CURRENT-PILE)
                   ) -->
                       (<TRY-TO-FIRE> (PICK-A-PILE))]

Type 2 Meta-rules:

 [(<WANT-INACTIVE>
              (FORGET-REMEMBERED-PILE)
                   ) -->
                       (<TRY-TO-FIRE> (FORGET-REMEMBERED-PILE))]
 [(<WANT-INACTIVE>
              (CLOSE-EYES
               OBJECT-IN-HAND-IN-VIEW
               FIND-COLOR-OF-PIECE
               FORGET-COLOR-OF-PIECE
               PIECE-HAS-CURRENT-COLOR
               PIECE-FITS-IN-PUZZLE
               FIT-PIECE-IN-PUZZLE
               PIECE-PUT-IN-PUZZLE
               PUT-OBJECT-IN-PILE
               REMEMBER-CURRENT-OBJECT
               REMEMBERED-OBJECT-IN-VIEW
               REMEMBERED-OBJECT-IN-HAND)
                   ) -->
                       (<TRY-TO-FIRE> (PIECE-PUT-IN-PUZZLE))]

Type 3 Meta-rules:

 [(<CURRENTLY-ACTIVE>
              (PUZZLE-IS-FINISHED
               THERE-ARE-NO-PILES)
                   ) -->
                       (<TRY-TO-FIRE> (PUZZLE-IS-FINISHED))]
 [(<CURRENTLY-ACTIVE>
              (PUZZLE-IS-FINISHED
               THERE-ARE-NO-PILES
               FORGET-REMEMBERED-PILE)
                   ) -->
                       (<TRY-TO-FIRE>
                                 (FORGET-REMEMBERED-PILE))]

Type 4 Meta-rules:

 [(<JUST-FIRED> (LOOK-AT-PIECE-IN-HEAP))
                     -->
                       (<TRY-TO-FIRE>
                                 (PICK-UP-OBJECT-IN-VIEW))]
 [(<JUST-FIRED> (PICK-UP-OBJECT-IN-VIEW))
                     -->
                       (<TRY-TO-FIRE>
                                 (PUT-OBJECT-IN-PILE
                                  MAKE-A-PILE))]
```

```
Try-to-fire(P'): procedure;
  TRYLIST: =
    TRYLIST ∪ (P' ∩ C); /* but duplicates remain */
  end;
```

The definitions are very simple and so the details of the implementation are not included.

The function Try-to-fire deposits a subset of its argument list of rule names in a master list (TRYLIST) maintained by the interpreter. All suggested rules must be active. The most often suggested rule is selected for execution; in the case of ties, preference is given to the rule name in the current simple unit, U.

ANALYZING SEQUENCES

When running a PS to a successful conclusion, a trace of the actions performed is generated. A series of these traces, called input trace sequences, are presented to a system which analyzes and produces CL/1 descriptions of them.

The exact form of an input sequence is $<<P_1\ P_2\ \ldots>>$ where P_i, called a trace unit, is of the form: $(P_i'\ (P_{i1}\ P_{i2}\ \ldots\ P_{it_i})\ (N_{i1}\ N_{i2}\ \ldots\ N_{ij_i})\ (M_{i1}\ M_{i2}))$. P_i' is the name of the rule which was applied on the i^{th} cycle of execution during the training session. $(P_{i1}\ P_{i2}\ \ldots\ P_{it_i})$ is the conflict set of rules on the i^{th} cycle. This set is used to calculate both the DNF expressions and the meta-rules.

The third component of the trace unit, $(N_{i1}\ N_{i2}\ \ldots\ N_{ij_i})$, is the set of unique integers (recency numbers, see [8]) associated with the instances of data elements which matched the pattern of the rule P_i'. Finally, $(M_{i1}\ M_{i2})$ is the range of unique integers associated with all of the data elements produced by the action portion of the rule P_i'. The last two pieces of information allow for the construction of a data flow graph, and consequently concatenations and permutations.

The trace sequence is an exact history of the execution of the program and is therefore already partially ordered by the back dominance relation: Every fired production in a trace unit within the sequence is preceded by those trace units which contain the fired production which produced data for it. This partial ordering simplifies much of the subsequent analysis of the trace.

Figure 3 is an example of a portion of a trace sequence produced by a run of the jigsaw puzzle PS.

The algorithms which construct CL/1 descriptions consist of several distinct passes over the input traces. Rather than having an enumerate and test character, these algorithms act by cutting and gluing subsequences of the trace, much like playing with construction paper and scissors. The details of the algorithms are presented in the next sections without formal definitions. Examples are used to demonstrate how they work.

PERMUTATION CONSTRUCTION

The input sequence is scanned and a unique number is assigned to each instance of a rule application in the trace (represented in the following as a subscript on the rule name). A directed acyclic graph (DAG) is then constructed with nodes (indexed by the unique numbers) corresponding to rule applications and edges representing the flow of data from one rule to another. This DAG is constructed in an obvious manner from the recency numbers. It is assumed that the last unit in the sequence signalled success of the problem solution and therefore its corresponding node is interpreted as the unique sink node of the DAG.

The DAG completely specifies all of the data dependencies between rules and therefore all of the possible execution sequences. In general, an arc from node v_i to node v_j indicates a concatenation of the subsequence associated with v_i (leading to and including v_i) with the unit associated with node v_j:

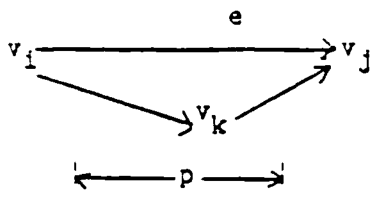
$$\begin{tikzcd} v_k:(P1_k) \arrow[r] & v_i:(P2_i) \arrow[r] & v_j:(P3_j) \end{tikzcd}$$

Figure 3: A Portion of an Input Trace Sequence

```
<<
(FORGET-CURRENT-PILE  (PUT-OBJECT-IN-PILE
         PUT-OBJECT-DOWN-ON-TABLE
         START-PUZZLE OBJECT-IN-HAND-IN-VIEW
         LOOK-AT-FIRST-IN-PILE
         LOOK-AT-OBJECT-ON-TABLE FIND-COLOR-OF-OBJECT
         REMEMBER-CURRENT-OBJECT PIECE-HAS-STRAIGHT-EDGE
         CLOSE-EYES REMEMBER-CURRENT-PILE)
     (510)(512 512))
(CONSIDER-ANOTHER-PILE (PICK-A-PILE MAKE-A-PILE
         PUT-OBJECT-DOWN-ON-TABLE
         START-PUZZLE OBJECT-IN-HAND-IN-VIEW
         LOOK-AT-OBJECT-ON-TABLE FIND-COLOR-OF-OBJECT
         REMEMBER-CURRENT-OBJECT PIECE-HAS-STRAIGHT-EDGE
         CLOSE-EYES)
    (505 512)(513 513))
(PICK-A-PILE (CONSIDER-ANOTHER-PILE MAKE-A-PILE
         PUT-OBJECT-DOWN-ON-TABLE
         START-PUZZLE OBJECT-IN-HAND-IN-VIEW
         LOOK-AT-OBJECT-ON-TABLE FIND-COLOR-OF-OBJECT
         REMEMBER-CURRENT-OBJECT PIECE-HAS-STRAIGHT-EDGE
         CLOSE-EYES)
     (512 513)(514 514))
(PUT-OBJECT-IN-PILE (LOOK-AT-FIRST-IN-PILE
          REMEMBER-CURRENT-PILE
          FORGET-CURRENT-PILE PUT-OBJECT-DOWN-ON-TABLE
          START-PUZZLE OBJECT-IN-HAND-IN-VIEW
          LOOK-AT-OBJECT-ON-TABLE FIND-COLOR-OF-OBJECT
          REMEMBER-CURRENT-OBJECT PIECE-HAS-STRAIGHT-EDGE
          CLOSE-EYES )
     (493 505 507 513 514)(515 518))
(FORGET-CURRENT-PILE (PICK-OBJECT-FROM-PILE
          LOOK-AT-NEXT-IN-PILE
          REMEMBER-CURRENT-PILE LOOK-AT-OBJECT-ON-TABLE
          FIND-COLOR-OF-OBJECT REMEMBER-CURRENT-OBJECT
          PIECE-HAS-STRAIGHT-EDGE CLOSE-EYES)
     (518)(519 519))
(CONSIDER-ANOTHER-PILE (PICK-A-PILE LOOK-AT-OBJECT-ON-TABLE
          REMEMBER-CURRENT-OBJECT CLOSE-EYES)
     (513 519)(520 520))
(PICK-A-PILE (CONSIDER-ANOTHER-PILE LOOK-AT-OBJECT-ON-TABLE
          REMEMBER-CURRENT-OBJECT CLOSE-EYES)
     (513 520)(521 521))
(LOOK-AT-FIRST-IN-PILE (LOOK-AT-OBJECT-ON-TABLE
          FIND-COLOR-OF-OBJECT REMEMBER-CURRENT-OBJECT
          CLOSE-EYES REMEMBER-CURRENT-PILE
          FORGET-CURRENT-PILE )
   (493 505 521)(522 522))
>>
```

would be represented in CL/1 as << P1 P2 P3>>.

The DAG is first cleansed of irrelevant arcs, called forward arcs. An arc e from node v_i to node v_j is a forward arc if there is some directed path p from v_i to v_j which does not include e as an intermediate edge. A forward arc specifies that node v_i must precede v_j and that node v_i precedes the nodes on path p. Since the nodes on path p enter v_j, they too must precede v_j.



Clearly, the forward arc e is superfluous and can be removed from the DAG, which prohibits the construction of irrelevant permutations. The resulting CL/1 description would be << p_i p_k p_j >>, where p_i, p_k and p_j are the rule names associated with the nodes.

The forward arc removal algorithm is due to R. Tarjan [22] and proceeds as follows:

Let DFSN(i) be a Depth-first Spanning Tree numbering assigned to node i beginning the numbering from the sink node of the graph. Let ND(i) be the sum of the numbers assigned to the immediate predecessors of node i. Then there is a path p from v_i to v_j iff $DFSN(i) \leq DFSN(j) \leq DFSN(i)+ND(i)$. Given v and the set of entering edges, it is easy to check from this inequality which edges are forward. For any two immediate predecessors, v_i and v_j, if the above inequality holds, then the edge from v_j to v can be deleted.

The construction of permutations is based on the observation that if a node has two or more entering edges then the subsequences associated with its predecessors can be permuted. The partial ordering makes this construction straightforward. Using a simple sequential scan, it is assured that the sequences associated with its predecessors have already been constructed when a node is being processed.

The permutations are further processed by first factoring out common leading subsequences (prefixes) and then applying the repetition detection procedure so that equivalences with other permutations could be more easily recognized.

Consider the following graph:

$$\begin{tikzcd}
v_1:(P1_1) \arrow[r] \arrow[d] & v_2:(P2_2) \arrow[r] & v_5:(P5_5) \\
v_3:(P3_3) \arrow[r] & v_4:(P3_4) \arrow[ur] &
\end{tikzcd}$$

The subsequences associated with nodes v_1, v_2, v_3 and v_4 are respectively, $<<v_1>>$, $<<v_1\ v_2>>$, $<<v_1\ v_3>>$, and $<<v_1\ v_3\ v_4>>$. When processing v_5 the description $<<(P.*\ <<v_1\ v_2>>\ <<v_1\ v_3\ v_4>>)\ v_5>>$ would be constructed (substituting the appropriate rule names for the v_i). Each of the subsequences begins with node v_1. By definition of permutation, this is equivalent to $<<v_1\ (P.*\ <<v_2>>\ <<v_3\ v_4>>)\ v_5>>$. This final form is less complex and more closely represents the control information in the DAG. Notice that if v_3 were instead $P2_6$ we could not factor out nodes v_2 and v_3 since they are distinct instances of applying rule P2 and could not be identified as the same. The final permutation can be further condensed by replacing the subsequence $<<v_3\ v_4>>$ by an operator specifying a repetition of P3.

Lastly, the node numbers assigned to the distinct rule names are then removed from the final description of the sequence.

<u>Repetition Routines</u>

The next process is the detection of repeating subsequences and replacement by repetition operators. Unlike the permutation detection problem, which is well-defined and solved by the algorithm described above, some sequences may have several equally acceptable descriptions. The algorithm that we use gives one description. It employs a helpful heuristic with a divide and conquer flavor.

The sequence is scanned to identify all single occurrences of rule names. All such occurrences cannot be part of a repetition and thus divide the sequence into shorter subsequences to which the same procedure can be applied recursively. Once a candidate subsequence is found (every atom in it occurs at least twice) a left to right scan is applied to find equivalent adjacent subsequences (beginning with the shorter) and to replace them with repetitions. The procedure stops when it considers subsequences whose lengths are greater than half the current length of the sequence. The maximum length of subsequences we have to consider decreases when repetitions are constructed. For example, given <<A B B C A B A B B>>, C occurs uniquely forcing the recursive calls on the subsequences it delimits. The first recursive call given <<A B B>> finds A to be unique and in turn recursively calls itself with <<B B>>. Here a repetition, (R.* <<B>>) is returned and concatenated with the A giving the description <<A (R.* <<B>>)>>. This is then returned and concatenated with C. This description is then concatenated with the returned value of the recursive call applied to <<A B A B B>>. The final description generated is <<A (R.* <<B>>) C (R.* <<A (R.* <<B>>) >>) >>.

The while and until components of the repetitions are computed from the conflict set of rules. A while condition is interpreted to be all of the active productions common to every entry of the repeating subsequence which are not active on exit from the subsequence. An until condition is the opposite; everything that is active on exit but which was never active previously.

The critical aspect of this approach is the equivalence of two sequences. Our interpretation is that the sequences must be lexicographically equal and aligned on a unit boundary. That is, one sequence cannot start within a repetition appearing at the end of the other. When two equivalent sequences are detected, a third sequence is constructed while merging the repetitions appearing in both.

For example,

```
  <<A (R.* <<B>> (W.* ()) (U.* (E))) C>>   and

  <<A B (R.* <<C>> (W.* (G)) (U.* ())))>>
```

are equivalent, and are merged to the sequence description

```
  <<A (R.* <<B>> (W.* ()) (U.* (E))
      (R.* <<C>> (W.* (G))(U.* ())) >>.
```

Although merging sequences has the potential of generating many more control statements than we might want, we have found that the while and until conditions will keep this to a manageable number (however, it is possible that empty conditions are computed). In fact, whenever a merge of two sequences occurs, we compute new while and until conditions for any repetitions that are merged by a disjunction of the individual conditions. This will mask out any erroneous iterations when the description is used in controlling a PS.

Notice there is no notion of similarity of subsequences (except in a very primitive sense) and so a slight perturbation in a repeating subsequence drastically effects the outcome of the analysis. There are many earlier attempts which focus on sequence extrapolation and prediction, but these reports focus on repeating patterns with constant periods [23]. The kind of analysis needed in that case is inadequate for inferring Regular Expressions. in particular, Kleene star. In this case, period size is variable when nesting of Kleene star is allowed leading to many alternative analyses. For instance, the sequence

```
  A B A A B A B
```

can be analyzed as

{ {A}* . {B} }* or { {ABA}* . {B} }

with no information indicating or suggesting the correct interpretation.

Alternative Subsequence Detection

The final process is the coalescing of several descriptions of a set of solution sequences into one general description. Our construction is similar to that of Winston [25]. From two or more descriptions of some object we identify the points of similarity and alternate the differences.

This problem can be viewed as being equivalent to the <u>Longest Common Subsequence Problem</u>. Since it is believed to be computationally intractable because of its NP-completeness in the case of three or more sequences, we do not intend to find the maximum points of similarity but rather a good heuristic approximation.

The set of descriptions is first ordered lexicographically to aid in discovering any equivalent ones. Then a <u>target</u> description is computed which is composed of a <u>series of subsequences</u> each common to all of the descriptions. This target identifies the points within a description that delimit smaller subsequences to be alternated. If the target is empty, a new target is computed. It proceeds by finding the <u>best</u> series of subsequences common to any two descriptions. <u>Best</u> is defined heuristically; more weight is given to permutations and repetitions than rule names. If the target in this case is still empty the descriptions are collected into one alternation with conditions computed as described below.

Using the final computed target, all descriptions are scanned for the first occurrence of the common target subsequence. The initial portions (prefixes) of the descriptions scanned are then collected in a set to which the entire procedure is recursively applied. The procedure continues in this way until the target is exhausted. A special tag is appended to the end of each description to force the algorithm to completion.

We demonstrate this process with an example. Given

    {<<B D E>> , <<B D F G>> , <<D F G>>}

the following events will occur: The target is computed to be

    {<<D>> <<T>>}

since D is common to all and T is the special dummy tag. Then we extract the prefixes

    <<B>>, <<B>> and << >>.

The entire routine is recursively applied to these. The ordering produces

    << >> and <<B>>

as the new set of descriptions. Since there is nothing common to both, the routine alternates them and returns

    <<(A.* (<< >>()) (<<B>>(B)))>>

to the top level. The sequence returned is concatenated with <<D>> to produce

    <<(A.* (<< >>()) (<<B>>(B))) D>>.

Now <<T>> is the target subsequence. The prefixes collected this time are

    <<E>>, <<F G>> and <<F G>>.

Again the recursion orders them getting

    <<E>> and <<F G>>.

No commonalities are detected so we return with

    <<(A.* (<<E>> (E)) (<<F G>> (F G)))>>.

The tag is removed from the sequence at the top level and the final description produced is

```
<<(A.* (<<>>())(<<B>>(B))) D (A.* (<<E>> (E))(<<F G>>(F G)))>>.
```

The computation of the conditions for the alternatives is very simple. For each alternate, we find the set of rule names true on entry to each. From each of these sets we remove any production name appearing in any other. Intuitively, the conditions specify the productions which are true when a particular alternative subsequence is entered but which are not true on entering any other.

Meta-rule Construction

Construction of the meta-rules is based on the transition of the PS environment when firing a rule. The first type of meta-rule is concerned with how rules are activated. From the trace we can determine which rules may activate other productions when fired by computing the set difference of the new conflict set with the old. Similarly, for the second type of meta-rule (how rules are deactivated) we compute the set difference of the old conflict set with the new. The Currently-active type meta-rule is constructed by copying the conflict set from which the rule was selected and the Just-fired type is constructed from the sequence of adjacent rules fired. Therefore, the meta-rules are constructed from fragments of the original solution sequence and contain much less information than the CL/1 description. However, as the reader will note below, when used in conjunction with the CL/1 description, both provide quite useful information.

Experiments

The jigsaw puzzle-solving PS consists of 36 productions containing an average of 3 data elements in the LHS, and was constructed in a few man hours. The experiment was performed on 6 puzzles, each containing 25 to 30 puzzle pieces (resulting in an average WM size of about 120 data elements during execution). One puzzle was used for training the system, and required over 400 cycles of production invocations to complete the puzzle. (A much shorter trace could have been produced, but much less interesting procedures would have been demonstrated or inferred.) The control information inferred from the trace consisted of 193 meta-rules and a CL/1 description containing 75 units (see [21] for the details). The inference program was able to construct procedures to sort the puzzle pieces into categories of those pieces with straight edges and those with common colors. For example, the (abbreviated) CL/1 description:

```
<<LOOK-AT-FIRST-IN-PILE

  (R.* (U.* (REMEMBERED-OBJECT-IN-VIEW))
      .<<LOOK-AT-NEXT-IN-PILE>>)

  (R.* (W.* (PIECE-HAS-STRAIGHT-EDGE))
       <<PICK-OBJECT-FROM-PILE

         FORGET-CURRENT-PILE

         CONSIDER-ANOTHER-PILE

         PICK-A-PILE

         PUT-OBJECT-IN-PILE

         FORGET-CURRENT-PILE

         CONSIDER-ANOTHER-PILE

         PICK-A-PILE

         LOOK-AT-FIRST-IN-PILE>>)  >>
```

which was inferred from the trace, scans one pile of indistinguished puzzle pieces removing those with straight edges and piling them separately. This description was generated from a portion of a solution trace which included the subsequence of figure 3. The portions of the CL/1 description which follow this consist of procedures for building the outside edges of the puzzle followed by similar procedures for adding pieces to the puzzle grouped according to common color.

It can be argued that the productions we have defined can be encoded by just a few rules (perhaps 2 or 3). In this situation, the control information for solving this problem is trivial. However, such a representation would be highly inflexible and admit only limited or narrow behavior. Further, many of the productions (in particular, the pile productions) are general enough to be used for other substantially different problems. It would seem possible then, that most of the control information we generate for these productions would be applicable in many different situations, and therefore would be of a more fundamentally general nature.

Of the six puzzles used in the experiment, three uncovered rather esoteric flaws in the CL/1 description. In each case the meta-rules were called upon to debug the error with interesting results.

In the first case, the correct rule selection was specified on each of the five calls to the meta-rules and the puzzle was correctly solved. The second puzzle required one meta-rule call, and was completed successfully. In third case, the meta-rules were called upon three times to solve a problem and succeeded twice. As Murphy's Law will have it, the third call to the meta-rules resulted in two choices from the conflict set having maximal and equal weight, the incorrect choice being selected. What is interesting is that in each situation, the meta-rules posted a small set of choices each containing the correct alternative. The rules we have designed are dependent only on the state of the PS program: the conflict set of rules and the dynamic behavior of the program during training. (This in effect, dynamically calculates the list of alternatives as opposed to static specification as, for example, PLANNER'S THUSE construct.) The effect of this limits the suggestions to those productions that were used in a solution, yet in many different but similar situations. If we were to relax the restriction of making irrevocable decisions and backtrack on the set of meta-rule suggestions, the system would have succeeded in all cases leaving little to chance.

Our system was also applied to a binary tree scanning program which demonstrated a difficulty with the CL/1 language. In a restricted form, CL/1 has the power of Regular Expressions and therefore lacks the power of a push-down automaton. The introduction of conditions testing rule applicability seemingly provides additional power but the resulting language remains difficult to classify in terms of the Chomsky hierarchy of languages. The binary tree example demonstrated the ability of CL/1 to scan balanced binary trees with a very sparse (yet appealing for its simplicity) PS program. (Six productions to traverse left, right and up, and to print the contents of a node.) However, constructing a simple CL/1 description to scan an arbitrary binary tree deterministically with the existing PS is impossible.

There are two directions in which we can proceed. We could redefine the PS to include rules to manipulate stacks. However, this would correspond to requiring the human expert to redefine the PS (not only additions, but rewriting existing rules) after discovering during training the desired control is not attainable. The alternative is to build a more powerful control language device. This seems the more satisfactory direction yet the required inference procedures would necessarily need to be more sophisticated.

In addition, the control specified by CL/1 is syntactic in nature, relying on the names of productions. In this regard, the ideas of Davis of semantic specification or <u>content-directed invocation</u> seem more appealing and much more flexible allowing for a wider variety of control used in perhaps new and interesting ways. It would appear then that a control language device with procedure calls and variables ranging over productions satisfying some condition applied to the contents of the rule would be desirable. It is not clear whether the kind of analysis we do with CL/1 could not provide the same ability by relaxing the way in which we use the resulting descriptions (i.e., select rule P1 or another rule <u>similar</u> to it). Future work and experimentation with the existing system may uncover some answers to these questions.

Conclusions

It is believed by many researchers that an important quality of intelligent behavior is the ability to improve performance with experience, and that generalizing a concept is a critical aspect of learning. In CL/1, a form of generalization occurs when a control operation is inferred from a successful trace. Although quite restricted in scope, CL/1 shows how a system might learn procedures, a form of learning which is believed to be very important. Although there are a number of very difficult technical problems, it seems to us that with more powerful pattern recognition techniques and more powerful generalizations of control statements, this approach could be very fruitful.

With less ambitious designs, CL/1 can be viewed as a programming aid for the designer and implementer of a large AI problem-solving knowledge base. One of the most error prone and difficult tasks in the development of an AI problem-solving system is the fine-tuning of the system with heuristic controls to minimize search times through a large data base of facts. The CL/1 approach might be useful in fine-tuning a declarative knowledge base as opposed to hand-compiling control elements to effect competent performance in such a system. This allows control information to be treated in a separate knowledge base independent of the declarative form.

The power of the descriptions produced is limited by both the expressive power of the CL/1 language, and the level of sophistication of the pattern recognition algorithms that have been developed. For example, during repetition detection no notion of similar subsequence is used; furthermore, it is not possible for alternation to appear within repetitions. Despite this, the algorithms are powerful enough to detect interesting patterns and subsequently interesting heuristics.

Our experiments suggest that this approach will have the best chance of success when the encoding of the knowledge base of the problem domain is such that on any execution cycle a small number of productions and only one instantiation of each production is applicable. For example, the initial segment of the CL/1 description for the jigsaw puzzle problem specifies repeatedly picking a piece from the heap and placing it in a pile. In actuality, the number of instantiations of the LOOK-AT-OBJECT-ON-TABLE production during this cycle is equal to the number of pieces currently on the table. The CL/1 description does not specify which instantiation to choose; but only the name of the production to choose. This suggests one way in which CL/1 could evolve: the execution traces should include not only productions but also instantiations of productions. The control language would therefore necessarily include mechanisms for semantic specification of control. Furthermore, the segmentation of the trace sequence into subsequences (or subproblems) specified by the human expert is not explicit in the final CL/1 description. Therefore, contextual or goal information should be included. However, recognizing patterns in such sequences is a much more difficult problem. The approach outlined in this paper is viewed as an initial step in the understanding of this more general problem.

The meta-rule construct we have defined seems quite effective. Although limited in scope, the meta-rules contain quite accurate and useful control information. The suggestion of a single rule in the RHS of a meta-rule effects a limited or local modification. More substantial statements of control can be effected by allowing arbitrary CL/1 descriptions in the RHS. However, a correspondingly more sophisticated analysis would be required. With more powerful meta-rules, the correctness of the complete CL/1 description would be less critical; we might then hope to get successful performance with meta-rules alone. However, this would probably require more understanding of some difficult context and control problems.

List of Figures

REFERENCES

(1) Abrahams, P., Machine Verification of Mathematical Proofs, Sc.D. Thesis, M.I.T., 1963.

(2) Angluin, D., An Application of the Theory of Computational Complexity to the Study of Inductive Inference, Ph.D., Thesis, U. Calif., Berkeley, 1976.

(3) Angluin, D., Finding Patterns Common to a Set of Strings, SIGACT Proc. Symp. on Theory of Comput., 1979.

(4) Bierman, A., On the Inference of Turning Machines from Sample Computations, Artif. Intell., 3, 1975.

(5) Davis, R., Applications of Meta Level Knowledge to The Construction, Maintenance and Use of Large Knowledge Bases, Ph.D. Thesis, Stanford U., 1976.

(6) Davis, R., and King, J., An Overview of Production Systems, Stanford U., AI Lab Mem., AIM-271, 1975.

(7) Fikes, R., Hart, P., and Nilsson, N., Learning and Executing Generalized Robot Plans, Artif. Intell., 3, 1972.

(8) Forgy, C., and McDermott, J., the OPS2 Reference Manual, CMU, Dept. of Comp. Sci., 1977.

(9) Gelernter, H., Realization of a Geometry Theorem-Proving Machine, Proc., Intern. Conf. Inform. Proc., UNESCO, House, Paris, 1959.

(10) Georgeff, M.D., A Framework for Control in Production Systems: Proc. IJCAI 6, Tokyo, 1979.

(11) Goldstien, I.P., and Grimson, E., Annotated Production Systems: A Model for Skill Acquisition, Proc. IJCAI 5, 1977.

(12) Kolmogorov, A., Three Approaches to the Quantitative Definition of Information, Problems in Information Transmission, 1, 1965.

(13) McDermott, J., and Forgy, C., Production System Conflict Resolution Strategies, CMU, Dept. of Comp. Sci., 1976.

(14) Newell, A., Production Systems: Models of Control Structures, in Visual Information Processing. W. Chase (ed.), Academic Press, 1973.

(15) Rychener, M.D., A Semantic Network of Production Rules in a System for Describing Computer Structures, CMU Tech Report, CS-79-130, 1979.

(16) Rychener, M.D., Control Requirements for the Design of Production System Architectures, SIGPLAN Notices, 12-8, 1977.

(17) Salomaa, A., Formal Languages. Academic Press, 1973.

(18) Sickel, S., A Search Technique for Clause Interconnectivity Graphs, IEEE Trans. on Computers, Special issue on Autom. Theorem Proving, 1976.

(19) SIGART Newsletter 70, Special Issue on Knowledge Representation, Brachman and Smith (eds.), 1980.

(20) Solomonoff, R., A Formal Theory of Inductive Inference, Information and Control, 7, 1964.

(21) Stolfo, S.J., Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces, Ph.D. Thesis, NYU, 1979.

(22) Tarjan, R., Finding Dominators in Directed Graphs, SIAM J.Comput., 3, 1974.

(23) Waterman, D.A., Serial Pattern Acquisition: A Production System Approach, in Computer Oriented Learning Processes, J.Simon (ed.)

(24) Winograd, T., Frame Representation and the Declarative/Procedural Controversy, in Representation and Understanding, Bobrow and Colins (eds.), Academic Press, 1975.

(25) Winston, P., Learning Structural Descriptions from Examples, MIT, MAC Tech Report-76, 1976.

(26) Zisman, M.D., Use of Production Systems For Modelling Asynchronous, Concurrent Processes, in Pattern-Directed Inference Systems, Waterman and Hayes-Roth (eds.), Academic Press, 1978.