

LEARNING TO USE ANALOGIES

John McDermott
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract. In order for a system to learn how to do new tasks, it must be capable of assimilation and accommodation. The assimilation capability enables a system to relate unfamiliar situations to situations that it knows about. Through assimilation, an unfamiliar situation is transformed, for a time, into a familiar situation. The accommodation capability enables a system to make such transformations permanent. ANA, the system described in this paper, is a production system that is capable of both assimilation and accommodation. Initially, ANA has a few methods for accomplishing a variety of simple tasks. When it is given a not too unfamiliar task, it performs that task by analogy with one of the tasks it has a method for. When it accomplishes a new task (and typically this happens only after the method has been extended to handle problems that it was not designed to cope with), it stores the knowledge of how it did the task. If ANA is subsequently faced with the same task, it recognizes the task and performs it using the knowledge previously gained.¹

1. INTRODUCTION

Any AI system that hopes to amount to much must have two capabilities. First, it must be capable of **assimilation**; it must be able to bring to bear whatever knowledge it has that is relevant to an unfamiliar task -- even though that knowledge was acquired in a variety of unrelated contexts. Second, it must be capable of **accommodation**; it must be able to augment and modify its knowledge so that unfamiliar tasks become familiar. The work described in this paper shows one way in which these two capabilities can, in a modest way, be realized.²

The system described, ANA, has limited knowledge about how to function in a simple environment. ANA is a production system; its productions are organized as a set of methods. The particular (and only) assimilation and accommodation strategy that ANA employs is to use these methods analogically. When given an unfamiliar task, it maps the description of a task for which it has a method into the description of that unfamiliar task. As it uses its method, instead of executing the actions prescribed by the method, it executes the actions dictated by the mapping. Whenever ANA uses a method successfully on an unfamiliar task, it builds a production that associates that method with the description of the

task (thereby making the unfamiliar task familiar). If in the course of doing a task the analogy breaks down, ANA attempts to patch the method; if it finds a patch, it builds a production that associates the patch with whatever caused the breakdown.

2. SOME CONTEXT

The production system architecture used to implement ANA is called OPS2 [2, 4, 7]. An OPS2 production system consists of a collection of productions held in **production memory** and a collection of data elements held in **working memory**. A production is a conditional statement composed of condition elements and action elements. Condition elements are templates; when each can be matched by an element in working memory, the production containing them is said to be instantiated. The production system interpreter operates within a control framework called the **recognize-act cycle**. In recognition, it finds all instantiations, and in act, executes the action elements of one of them. The recognize-act cycle is repeated until either no production can be instantiated or an action element explicitly stops the processing. Recognition can be divided into match and conflict resolution. In match, the interpreter finds the conflict set, the set of all instantiations of productions that are satisfied on the current cycle; OPS2 is implemented in such a way that the time needed to compute the conflict set is essentially independent of the size of production memory (see [1]). In conflict resolution, the interpreter selects (on the basis of a few simple rules) one instantiation to execute (see [5]). The actions that can be performed include adding elements to and deleting elements from working memory and building new productions composed of elements in working memory.

¹This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, and monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

²For a general discussion of the issues that face anyone designing a system with assimilation and accommodation capabilities, see [8].

To provide ANA with the capability of performing a variety of tasks, its knowledge is represented as a set of methods. Each method contains the knowledge that ANA needs in order to achieve some goal. Since part of this knowledge is the knowledge of subgoals that have to be achieved, ANA's knowledge is organized, though only implicitly, as a hierarchy of methods. The productions that comprise each method have condition sides with a quite simple form. This form is perhaps most easily described in terms of the type of data element that each condition can match. A data element contains information about a goal or about some feature of ANA's environment. In addition, each data element contains a subelement, which I will refer to as the marker, that specifies, among other things, the element's type (see [8]). There are four types of data elements: **goals, constraints, percepts, and concepts.**

A data element marked "goal" contains two subelements (plus a marker). One of these is a pointer to the set of constraints on the action to be performed; the other is a pointer to the set of constraints on the object to be acted on. A data element marked "constraint" contains three subelements (plus a marker); constraints are attribute-name-value triples. A goal element and the two sets of constraints it points to are collectively a goal description. When ANA attempts a task, it finds an object that fits the description stipulated by the object constraints and performs the type of action stipulated by the constraint containing the type of the action. The other action constraints specify the expected effect of performing that type of action on the class of objects described. Figure 2-1 displays a goal description. The stipulated action type is paint. The object to be painted is a yellow table in a location whose label is L32. The expected effect is a change to the surface of the object; specifically, the color of the object should be red once the goal is achieved.

```
(act*1 object*2 (goal not-achieved 3))

(type act*1 (constraint given 4) paint)
(effect act*1 (constraint given 4) effect*5 surface)
(color effect*5 (constraint given 4) red)
(type object*2 (constraint given 4) table)
(color object*2 (constraint given 4) yellow)
(location object*2 (constraint given 4) location*6)
(label location*6 (constraint given 4) L32)
```

Figure 1: A goal and a set of constraints

A data element marked "percept" contains information about one of the objects in ANA's environment. In ANA's world, an object is just a bundle of percepts (a set of attribute-value pairs). ANA "sees" an object whenever a production containing the operator @scan fires. @scan takes a (partial) description (a set of attribute-value pairs) as its argument and searches the environment for an object matching that description. If

it finds such an object, it deposits a set of elements, each of which is marked "percept", in working memory; each of these data elements corresponds to one of the attribute-value pairs of the object found. Clearly, in order for ANA to be able to distinguish among objects, these percepts must somehow be linked. Thus, @scan puts a "token-name", as well as an attribute and a value, in each percept that it deposits in working memory.

A data element marked "concept" can be thought of as a processed percept. When ANA looks at something, it does so for a reason -- ie, to find out something about the object. For each percept that contains information that it cares about, it asserts an identical element marked "concept". In addition, it asserts an element, also marked "concept", that relates the set of constraints on the object with the object's token-name. The concept marker has several uses: (1) It enables ANA to focus its attention on particular features of an object. (2) ANA can pretend that an object has a value that it in fact does not have by asserting a concept containing that value. (3) ANA can use concepts to store non-perceptual knowledge about an object. Figure 2-2 shows a collection of percepts (the attribute-value pairs that comprise the object that ANA thinks of as table*1); the figure also displays two concepts. When ANA is given a task, the description of the object to be operated on may, but need not, uniquely specify an object. In order to do the task, ANA must find an object that matches the description given. ANA's productions distinguish between information that constrains the selection of an object and information about a particular object. Elements containing the first sort of information are marked "constraint", while elements containing the second sort are marked "concept". If ANA were given the task shown in Figure 2-1, and if it looked for a chair whose color was blue in order to determine its location, then ANA would assert the concepts shown in Figure 2-2. The concepts would indicate to ANA that the table it refers to as table*1 is the one to be operated on.

```
(type table*1 (percept true 7) table)
(color table*1 (percept true 7) yellow)
(weight table*1 (percept true 7) light)
(location table*1 (percept true 7) L*32)
(position table*1 (percept true 7) (1))

(location table*1 (concept true 8) L*32)
(token object*2 (concept true 8) table*1)
```

Figure 2: Some percepts and concepts

As is evident from Figures 2-1 and 2-2, an element's marker specifies more than just the type of the element. It contains information about the status of the element. A constraint, for example, may be "given", "not-given", or "possible". The marker also contains information indicating the element's recency.

3. ANA'S TASK ENVIRONMENT

ANA's task is to manage a paint shop. ANA operates a machine that sits in the middle of the shop. The only things in the shop other than the machine are a variety of paintable objects. The shop and the things in it are all abstractions of real objects; they are represented as collections of attribute-value pairs. The shop is a 3 by 6 matrix; each square in the matrix is a stack with type location, one of 18 positions ((1 1) - (3 6)), a label (L11 - L36) and a composition (the stack of objects that occupy it). Each of the objects in the shop have a type (eg, chair, box, desk), a location, a position in the location, a color, a weight, and a state (clean or dirty). The shop and its current contents are shown in Figure 3-1.

ANA has five operations that it can perform on the objects in the shop. One of its operators, @spray, starts the machine. Three operators, @carry, @push, and @cart, enable it to move objects from one location to another. ANA's other operator, @scan, as I mentioned, enables it to see what is in the shop. In keeping with the somewhat artificial nature of the shop and its contents, there are some artificial constraints on these operators. Which of the move operators is applicable is determined by the weight of the object to be moved. @carry can be used only with light objects, @push only with heavy objects, and @cart only with really heavy objects; @carry cannot be applied to an object unless it is the top object in a stack, and neither @push nor @cart can be applied unless the object is the only object in a location. If an object is carried to a location that already contains four objects, it will fall onto an adjacent location; if an object is pushed or carted to a

location that already contains one or more objects, it will end up in an adjacent location. In order to paint an object, ANA must move it to location L23 and must put a paint can on top of the machine (in location L24); the operator, @spray, will modify the color of the object in L23 and cause it to be output in location L34. If there is not exactly one object in L23, @spray does nothing; if there is no sprayable substance on top of the machine, the object in L23 will be output to L34 unchanged. The move operators all take the "token-name" of the object to be moved and the "token-name" of the location to which it is to be moved as their arguments. Thus these operators all presuppose a prior @scan.

ANA's initial task-specific knowledge consists of six methods. ANA has a method for painting tables whose location is L32 red. ANA also has a method for clearing off the top of desks that are in L32. ANA's other methods are all methods for transporting objects. One of these methods enables it to move boxes that are in L25 to some unspecified location. A second method enables ANA to carry tables in L32 to L23. A third method enables ANA to carry cabinets in L11 to L16; this method assumes that the position of the cabinet in the stack is known (and is something other than top; and so calls the cleartop method as a submethod. The fourth method enables ANA to cart cabinets in L31 to L23.

Two of the tasks that ANA has been given will be used as examples in the following sections. The two tasks are: (1) "paint the blue chair in L21 red and then move it to L35", and (2) "wash the thing in L12". ANA's initial methods do not enable it to do either task directly; in both cases ANA must perform the tasks by analogy.

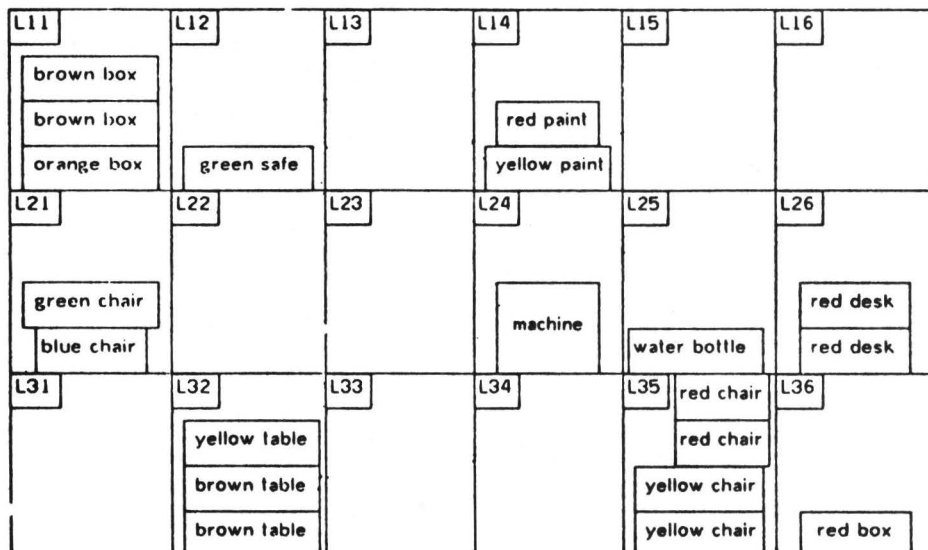


Figure 3-1: ANA's paint shop

4. ANA'S LEARNING STRATEGY

To do the tasks described in the previous section, ANA has to be able to assimilate new knowledge. Since the few methods that it has for getting things done in the shop are not (for the most part) the methods needed to do these tasks, it must find a way to apply the knowledge it has to the unfamiliar tasks. ANA's strategy is to find a method that is adequate for a related task and assume it will be adequate for the new task. Since this assumption will often lead it to do the wrong thing, it must have ways of correcting its errorful actions. ANA's solution is to watch for signs that it is not accomplishing its task. Three kinds of problems can arise. If the method it is using generates a subgoal, ANA tries to determine whether that goal is appropriate within the context of its current task; if ANA decides it is not, it substitutes an appropriate goal. If ANA sees that it has achieved a result other than the one it intended to achieve, it assumes that the method it is using does not take into account one or more constraints imposed by the particular task at hand; in this case it tries to determine what the additional constraints are and then attempts the task again keeping those constraints in mind. If ANA sees that it is unable to achieve a result because of some constraint imposed by the method it is using, it checks with its task master to see if that constraint is necessary; if the task master indicates that it is not necessary, ANA ignores the constraint.

The first time ANA does an unfamiliar task it spends a great deal of time selecting methods to use, establishing and extending mappings, and recovering from errors that it falls into. On subsequent occasions, ANA can do the tasks more directly. Roughly speaking, all that ANA does is store worked-out analogies. It associates each newly familiar goal description with a method that can achieve that goal analogically. ANA does not try to generalize; thus the only goal descriptions that it recognizes (ie, does not have to assimilate) are goal descriptions that it has previously encountered. Though this limitation is significant, it is not as severe as it perhaps sounds. In the first place, as ANA engages in some task, it typically generates a number of subtasks for itself. The knowledge that it stores about how to perform each of these subtasks is accessible in any context -- ie, not just in the context of the particular task in which the knowledge was gained. Thus if ANA is given an unfamiliar task, though it will have to use its analogy mechanism in order to do the task, it may turn out that some of the subtasks that it generates in the course of doing that task are identical to subtasks that it has already learned how to do. Whenever this happens, ANA can use the knowledge that it previously acquired. Secondly, although ANA learns how to do a specific task, what this means is that it learns what to do given a particular set of constraints on the action and the object to be acted on. If ANA is given a task that includes those constraints, but others besides, it

can use its acquired knowledge. Of course the fact that there are additional constraints may mean that ANA's knowledge will be inadequate for the task; but if so, ANA can make use of its error recovery methods to further refine its knowledge.³

In this section I will give a general description of ANA's learning strategy. I will first describe how ANA sets up analogies and will then describe how it recovers when an analogy breaks down.⁴

4.1 Setting Up Analogies

When ANA is given a task that it knows how to do (ie, a task for which it has a method), it needs no special mechanism to make contact with that method; the productions comprising the method are satisfied by elements in working memory and fire. When ANA is given an unfamiliar task, however, it must select an appropriate method and then assert elements that will satisfy that method's productions. To enable ANA to make contact with an appropriate method, each of its methods has associated with it a method description production. Each method description production contains two condition elements that specify the type of action and object with which its method is equipped to deal. It may contain any number of additional condition elements specifying constraints on the action and object. When given a unfamiliar task, ANA first generates action types and object types that are related to the action and object types associated with its current task. The knowledge it uses is represented as a tree of types; each node of the tree is defined by a set of productions. One of the productions has as one of its action elements an element indicating the supertype of that type; each of the other productions has as one of its action elements an element indicating an instance of that type. After some number of these isa net productions fire, one of the method description productions will be satisfied and will fire. The action part of the production contains the set of constraints that the associated method is familiar with. These constraints are marked "constraint possible" rather than "constraint given" to indicate their provisional nature. A method description production simply proposes its method; before ANA uses the method it evaluates the method's adequacy.

³Anyone familiar with HACKER [9] will have noticed that HACKER and ANA have similar aspirations. Both systems start out with a limited amount of knowledge and try to build on that knowledge in such a way that they become increasingly better able to accomplish unfamiliar tasks. But they have rather different building strategies. When HACKER encounters an inadequacy in one of its procedures, it examines the code; when it finds the bug, it rewrites the procedure. ANA is more of a hacker. When ANA encounters an inadequacy in one of its methods it determines what behavior is appropriate and associates that behavior with a description of its current context. Each of ANA's fixes is pre-eminently a patch.

⁴ANA's capabilities are described in considerably more detail in [3].

The first step in this evaluation involves setting up a mapping between the description of its actual goal and the pseudo-goal-description generated by the method description production. ANA finds all of the attributes in the description of the actual task ("constraint given"s) for which there are corresponding attributes ("constraint possible"s) asserted by the method description production. For each such pair of constraints, ANA asserts an element that indicates that the method's value (hereafter called the "pseudo-value") is to be mapped into the stipulated value.

After these mappings are established, the evaluation of the proposed method begins in earnest. ANA first checks to determine if there is any reason to think that the method will not work for the task at hand. Initially, it has no reason to think badly of any method. But as we will see in more detail later, when it encounters a problem as it tries to use a method, it builds a production that associates with the type of action that it is attempting an indication of the cause of the problem. If, for example, ANA attempts to carry an object that is buried down in the middle of some stack, its initial attempt will fail. But ANA will recognize that the cause of the problem is the position of the object in the stack, and will associate a pre-condition with the action type "carry", ie, that the object to be carried must not have anything on top of it. The problems that arise can be distinguished on the basis of how likely it is that they can be overcome. ANA divides problems into two types: (1) problems that arise because of some mutable attribute of an object, and (2) problems that arise because of an immutable attribute of an object. When a pre-condition is violated by an immutable attribute of an object, ANA concludes that the method it is considering is unlikely to work. When there are no immutable violators, but a pre-condition is violated by a mutable attribute, ANA concludes that the method will probably work, but only if a subgoal that can satisfy the pre-condition is achieved.

After ANA has evaluated a method, it typically returns to its isa nets and generates additional types with the hope that a better method will turn up. How long it spends searching depends on how good its current best candidate is; it spends more time if the current candidate falls into the "unlikely to work" category than if it falls into the "will probably work" category.

Once ANA has selected a method to use, it must somehow change its conceptual world in a way that makes it possible for it to use the method. The problem is that all of the productions comprising each of its methods are sensitive only to those constraints that are marked "constraint given". Since the evaluation stage results only in a set of constraints marked "constraint possible", ANA still cannot do anything. But the solution is straight-forward. ANA asserts, for each constraint marked "constraint possible", an element that is identical to the "constraint possible" element except that it is marked "constraint given". Now in ANA's working memory in place of the single set of constraints

stipulated in the description of the actual task, there are three sets of constraints: (1) the original set (each marked "constraint given"); (2) the set generated during the evaluation stage (each marked "constraint possible" and containing the pseudo-values expected by the method, rather than the stipulated values); and (3) a set that will match the productions in the method selected (each marked "constraint given", but containing the same pseudo-values as the set marked "constraint possible"). As simple as this device is, it is sufficient to enable ANA to make effective use of the selected method. Whenever ANA runs into difficulties, it has all the information that it needs in order to distinguish between what it is actually trying to do and what it is pretending to do. Since ANA is always implicitly aware of the difference between what it is trying to do (its actual goal) and what it is pretending to do (its pseudo-goal), whenever its pretending results in an action that is inconsistent with its actual goal, it can use this difference to figure out how to get back on the right path.

Next ANA prepares for one type of difficulty that might arise as it uses the method analogically -- the "unexpected result". ANA's method description productions typically include one or more condition elements that indicate the expected effect of the method. As ANA prepares to use the method, it builds a production that watches for the violation of this expectation for the particular task at hand. The method description production for ANA's "paint table" method, for example, indicates that the expected effect is that the color of the table will become red. If ANA is given the task of painting some object yellow, it wants to be sure that the effect actually achieved is that the table becomes yellow. So ANA builds a production that keeps its eye, so to speak, on the object; if after the object is painted it is not yellow, this production can fire.

ANA is now ready to use the method it has selected. However, if it is to avoid having to set up the same analogy on subsequent occasions, it must store the knowledge it has gained in a readily accessible form. Thus, just before it actually attempts an unfamiliar task, ANA builds a production that will "set the stage" if it is ever again asked to do the task it is about to try. In order for this production to fire at the appropriate times, it must be sensitive to the set of stipulated constraints that comprise ANA's current goal. Thus, ANA generates a list containing the element in working memory that points to the constraints on this goal and each associated element, marked "constraint given" for which there is no corresponding element marked "constraint possible". The pointer in each of these elements is replaced by a variable, and the list of elements becomes the conditional part of the production that ANA will build. The action side of the production must make contact with the method that has been selected for the task; consequently, ANA generates a list containing each "constraint given"/"constraint possible" pair linked to the current goal, plus each of the map

elements associated with these constraints. After the pointer in each of these elements is replaced by a variable, the list of elements becomes the action part of the production that ANA will build.

At this point, ANA could simply build the production, but there is a problem. Since it has not yet tried to use the method it has selected, it has no idea whether the method will work. If ANA were to build the production before trying the method, and the method turned out not to work for the current task, ANA would have a piece of faulty knowledge. ANA's solution is to build a production whose condition side contains just one element -- an element that will match the current goal element when that element is marked "goal achieved". If this production fires, the action side builds the production described above.

4.2 Using Analogies

As I indicated above, there are three ways in which ANA can get into trouble as it tries to use a method analogically: (1) The method may generate a subgoal that is inappropriate in the context of its current task. (2) The method may be under-specified (ie, insufficiently constrained), and so ANA ends up operating on (or trying to operate on) some object in an inappropriate way. (3) The method may be over-specified (overly constrained), and so ANA achieves a goal, but does not know it.

Whenever a subgoal is generated by a method, a new (distinct) description of the action and its expected effect is asserted. But the description of the object to be acted on may or may not be new. If it is not new, ANA simply checks the description of the action to see if it contains any new constraints. ANA has a production whose purpose is to watch for any constraint whose value is the pseudo-value of some map element. Whenever such an element is asserted, this production fires; the result is two additional assertions. One is identical to the assertion that triggered the production, except it is marked "constraint possible"; the other is an assertion, marked "constraint given", whose value is the stipulated value of the map element. If the object to be acted on is new, ANA tries to determine whether it is an appropriate object for the task at hand. If ANA decides that it is not appropriate, it replaces the object description asserted by the method with the description of an appropriate object. For example, ANA has a method for painting tables that generates the subgoal of putting a paint can on top of its "paint machine". If ANA is given the task of washing an object, it decides that it should substitute water for paint.

ANA has two ways of determining that it is using an under-specified method. One is to wait until one of its operations on an object in its environment fails. When an operator (eg, @carry, @cart) fails, an element marked "action failed" is deposited in working memory; this element identifies the cause of the failure. If, for example, @carry fails because the object to be carried

is not the top object in a location, the "action failed" element will indicate that the offending attribute is "position" and that carry will work only if the object is at the top of the stack. ANA has a production that watches for an "action failed" element to be deposited in working memory. When such an element is asserted, this production fires and generates the goal of recovering from the error. The first thing that the error recovery method does is check to determine if the offending attribute is mutable or immutable. Since ANA responds differently depending on whether the offending attribute is mutable or immutable, we will consider each case separately.

We have already seen what happens in the mutable case. If an operator fails because of a mutable attribute of some object, ANA builds a production associating the current action type with that attribute and the value it must have. If ANA knows how to change the attribute, it generates a subgoal to do so; if it does not know, it asks the task master. This fix is sufficient to enable ANA to deal effectively with the problem. In the future, if ANA considers a method whose action type is associated with the necessary, but mutable, value of some attribute, and if the object does not have that value, ANA knows that to use the method it is considering, it must generate a subgoal to satisfy this pre-condition.

The immutable case is more complex. If the problem is due to some feature of the environment that cannot be changed, ANA must find a new way of dealing with the situation. What this means, in ANA's simple world, is that ANA must use a different operator to effect whatever change it desires. When ANA finds itself in such a situation, it builds a production whose condition side is the same as the condition side of the production that actually operates on the environment; this production generates the subgoal of finding the value of that attribute of the object that is relevant to the operator. It builds a second production whose condition side is identical to the first except that it includes a condition element that is satisfied if the value of that attribute is different from the value required by the operator; the action side of this production, executes a different operator. The reason for making the condition side of this production a special case of the original production is that ANA wants the original production to fire unless the information that is obtained about the object to be operated on indicates that the original production's operator is inappropriate.

ANA's other way of determining that it is using an under-specified method has already been discussed. When ANA decides to use a method analogously, it builds a set of productions to watch for unexpected results. Whenever ANA operates on an object and the result is different from that expected by one of these productions, it will fire. When it does, ANA first asks how to undo the unwanted result and then asks for an indication of what went wrong. ANA expects to be told that the value of one of the attributes of one of the

objects in its environment is wrong and expects to be told what the value should be. ANA then treats this information in exactly the same way that it treats the information contained in an element marked "action failed".

ANA's method for determining whether the method it is using is over-specified is to ask. It has a production that waits until the goal of finding some object is generated, and then looks to see if the values of any of the expected effect constraints on that object differ from the values of the corresponding percepts. If there is a discrepancy, the production fires. ANA asks the task master whether the value of the attribute has to be the stipulated value. If the task master answers "yes", then ANA keeps looking until it finds an object with the stipulated value. If the task master answers "no", ANA assumes that the method it is using is over-specified, and asserts an element that indicates that the object that it has found has the stipulated value (even though it does not). In order to produce a permanent fix, ANA builds a production that will assert that element whenever it is given an object to find in the context of the current action type.

5. TWO SAMPLE TASKS

In this section, ANA's behavior on the two sample tasks is described. Taken together, the two tasks provide examples of most of ANA's capabilities.

5.1 The Chair Painting Task

When ANA is given the task of painting the blue chair in L21 red, since it has no method that is directly applicable, it must make contact with an appropriate method. After a few of its isa net productions fire, the method description production for painting tables is satisfied and fires. The action type "paint" is mapped into itself, the description of the expected effect of painting is mapped into itself, the object type "table" is mapped into "chair", and the description of the location of the table (that it is in L32) is mapped into the description of the location of the chair (that it is in L21). ANA then looks for other candidate methods. Since it finds none, it selects the "paint table" method as the method to use. For each element asserted by the method description production marked "constraint possible", it asserts a corresponding element marked "constraint given". It then builds a production that will fire if the chair, after being painted, is not red, and it builds a production which will build a production that associates the task of painting a chair in L21 with the "paint table" method if that method is used successfully. The first action of the "paint table" method is to assert

the subgoal of finding the table (chair) to be painted.⁵ Once the chair is found (by means of @scan), the subgoal of carrying the table (chair) to L23 is asserted. Since ANA has a method for carrying tables from L32 to L23, and since the pseudo-goal generated by the paint method is to carry a table from L32 to L23, ANA does not need to search for a method to use to accomplish the carrying goal.

The first action of the "carry table" method is to assert the subgoal of finding the table to be carried; again the blue chair in L21 is found. Then the method asserts the subgoal of finding the location to which the table (chair) is to be carried. The goal description stipulates that this location is to be L23 and the expectation is that this location is empty. Since L23 is a new object (ie, is not mentioned in the initial description of the task to be performed), ANA has to decide whether for its current task a different location should be substituted for L23. It has no knowledge that leads it to think that a substitution should be made, so it accepts L23 as the location to which it should carry the table (chair). The next action is to use the @carry operator to move the table (chair) to L23. This operator is applied, but fails because the chair it tries to carry (the blue chair) has another chair on top of it. ANA knows that the position of an object in a stack is a mutable attribute. Thus it builds a production that associates with the action "carry" the information that in order for an object to be carried it must be at the top of a stack. Whenever ANA subsequently considers carrying an object, this production will fire. ANA knows that if an object must be at the top of a stack and is not, that it can remedy this by asserting a cleartop goal. Thus, after ANA builds the production it asserts the goal of clearing off the top of the blue chair.

When it asserts the cleartop goal, it finds that it has no immediately applicable method, so it uses its isa nets to make contact with an appropriate method. It selects its method of clearing the top of a desk in L32. This method generates the goal of moving the top object in the stack. Again, ANA has no method for moving a chair from L21, but it does have a method for moving a box from L25. It selects this method to use. One of the constraints on the "move box" method is that the expected effect of moving the box is that it end up in L33. When ANA uses the method, it sees that it could move the green chair to L22 and so asks the task master if it is necessary that the chair end up in L33. Since there is no reason in this case for the chair to be moved to L33 (ie, since the method is over-specified) the task master indicates that it is not necessary and ANA moves the chair to L22. Having achieved its move

⁵ANA's method for finding objects, unlike its other methods for operating on objects in the paint shop, is quite general. The "find" productions are aware of the "constraint given"/"constraint possible" distinction; if this were not the case, when "find" was invoked, ANA would be unable to determine whether to @scan for the actual object desired or the pseudo-object specified by the method being used.

goal, the production that ANA built to build a production that associates the task of moving a chair in L21 with its "move box" method fires. Since the clear top goal is also now satisfied, the production which builds a production that associates the task of clearing off the top of the chair in L21 with its "clear top desk" method also fires. Once the blue chair is at the top of the stack, the carry method again comes into play. This time, when the @carry operator is executed, it succeeds, and so the chair ends up in L23. Since the carry goal is satisfied, the production which builds a production that associates the task of carrying the chair with the "clear top desk" and "carry table" methods fires.

When the chair has been moved to L23, the "paint table" method asserts the goal of carrying the red paint can to L24. Since the red paint can is a new object, ANA has to decide whether it should substitute some other object for it. ANA knows that the purpose of a paint can is to contain paint. It also knows that the purpose of paint is for painting. Since painting is the stipulated action type for the task at hand, ANA assumes that the type of the object that it wants to carry is paint can. After deciding that the stipulated object type is in fact appropriate, it checks the other constraints on the object. Since it knows that the label on a paint can indicates the color of the paint inside the can, it checks whether the description of the expected effects of doing the painting make any mention of color. Since the expected effect of painting the box is that it will become red, ANA concludes that the red paint can is appropriate. Since ANA has no method for carrying a paint can, it selects its "carry table" method again and uses it successfully. When both the object to be painted and the paint can are in the appropriate locations, the "paint table" method executes the @spray operator, and the newly painted chair ends up in L34. Then the subgoal of carrying the paint can back to its original location is generated, and "carry table" is used successfully.

Now ANA must move the newly painted chair to L35. The "carry table" method is selected. After selecting the method, ANA (as usual) builds productions to watch for unexpected results. One of the productions built in this case watches to make sure that after the chair is carried, it is in location L35. In this particular case, since there are already four chairs in L35, when the chair is added to the top of the stack, the stack testers and the chair ends up in location L34. When the production that watches for this unexpected result fires, the task master is asked how to undo the unexpected result. Since the chair fell back to the spot where it was, and since everything else is unchanged, the task master indicates that nothing need be done. Then the task master is asked to indicate what went wrong. All he need say is that there are too many objects in L35; there have to be fewer than four. ANA then builds a production containing the information that in order to be able to carry something somewhere, the location that the object is being carried to must contain fewer than

four objects. After building the production, it reconsiders what method to use for the task. This time, when it considers carrying the chair, it realizes that in order to achieve its goal it must first achieve the subgoal of moving one of the objects in L35 somewhere else. After the chair at the top in L35 is moved out of the way, the "carry table" method is again applicable and succeeds in getting the chair to L35 for keeps.

5.2 The Safe Washing Task

When ANA is given the task of washing the thing in L12, it again selects its method for painting tables. This time, the action type "paint" is mapped into "wash", the description of the expected effect of painting (that the object painted will be red) is mapped into the expected effect of washing (that the object washed will be clean), the object type "table" is mapped into "thing", and the description of the location of the table (that it is in L32) is mapped into the description of the location of the thing (that it is in L12). When the "paint table" method generates the subgoal of carrying the table (thing) to L23, the "carry table" method discovers that the only thing in L12 is a safe and using one of its isa nets verifies that a safe is a thing. When it attempts to carry the safe to L23, @carry fails because the safe is not light. An "action failed" element is asserted; the production watching for such an element fires, generating the goal of recovering from the error. Since ANA knows that weight is immutable, it knows that it cannot fix the problem by modifying the state of the world. So it tries a different operator -- @push. It builds a production that temporarily masks (ie, that is selected in preference to) the production whose action side contains @carry. The function of the new production is to look to see whether the object to be carried is light. ANA then builds another production whose condition side is identical to the condition side of the production just built except that it contains an additional condition element that will match a percept whose attribute is "weight" and whose value is anything other than "light"; the action side of this second production contains the operator @push. Then, since ANA sees that the weight of the thing (ie, the safe) is "heavy" (ie, not "light"), the second production will fire; this time, ANA's attempt to move the safe is successful.

The "paint table" method next asserts the subgoal of carrying the red paint can to L24. After recalling that the purpose of a paint can is to contain paint and that paint is for painting, ANA notices that what it actually wants to do is wash. So it tries to recall an object that has the same relationship to washing as a paint can has to painting. Since it knows that water is what is used for washing and that water bottles contain water, it decides that it should substitute something of type water bottle for the paint can. It looks around the shop for a water bottle and when it finds one, assumes that it has all of the appropriate attributes and substitutes the perceived values of the water bottle for the corresponding values in the description of the paint can.

ANA carries the water bottle to L24, the "paint table" method executes *nspray*, and the task is done.

6. CONCLUSIONS

It should be evident by now that ANA has a somewhat cavalier attitude toward learning. Whenever it has to learn, it learns (at most) just enough to get by. When it is given an unfamiliar task, it tries to do the task by analogy; it has a few rather weak rules that enable it to select a method to use. It maps the description of the goal that the method can achieve into its actual goal; it does not consider at this point whether the mappings are plausible. It then attempts to use the method; it does not construct a plan (since the method serves as its plan). When the method succeeds in accomplishing a task, ANA builds a production that enables it to subsequently recognize that task and to remember the analogy; it does not create a new method. When an analogy breaks down, ANA patches the method it is using by building a production that will watch for signs of trouble and take steps to avoid it; ANA does not modify any of the productions comprising the method. Should ANA be doing some of these things that it does not do? Which of them would improve its performance and which would merely slow it down?

Given ANA's performance on the two tasks, one conclusion that seems warranted is that if a system is provided with a set of highly specific methods for performing a few tasks, and if at least one of these methods is almost adequate for each unfamiliar task the system will face, then an assimilation and accommodation strategy like the one ANA employs enables the system to learn to perform unfamiliar tasks without requiring it to know much about learning. If the methods are almost adequate, then the types of problems that the system can encounter are quite limited: (1) Subgoals that are generated by the method may not be appropriate for the unfamiliar task; so the system will have to learn what the analogous subgoals are. (2) The method being used can be under-specified; so the system will have to learn what the additional pre-conditions on the method are. (3) The method being used can be over-specified; so the system will have to learn what constraints to ignore. The mechanisms required to solve all three problems are quite simple.

ANA's strength, then, is that its learning mechanisms are simple, but effective -- at least for simple tasks. And since ANA recovers from method inadequacy by creating patches locally as particular problems arise, task complexity, of itself, presents no special difficulties. If ANA is given a complex task that can be decomposed into a set of subtasks for which it has almost adequate methods, ANA's learning mechanisms will enable it to patch those methods appropriately. The fact that ANA is so dependent on a store of almost adequate methods may appear to be a significant limitation. But these methods are highly specific and thus easily acquired. The knowledge embedded in each of ANA's task methods is just that knowledge which would be acquired if ANA

were to be led, step by step, through a particular task. Thus with some somewhat laborious training, ANA could acquire a store of methods sufficient to enable it to perform a wide variety of unfamiliar tasks. ANA does, however, have a serious weakness: its knowledge of how to select an appropriate (almost adequate) method is extremely limited. If ANA had a large number of methods from which to select, it would need more knowledge of the interrelationships among actions and among objects and more knowledge of how to determine the dimensions along which to compare tasks. If ANA had such knowledge, and if it had a large store of methods, its learning strategy could be effective in many non-toy domains.

ACKNOWLEDGEMENTS

The development of many of the ideas discussed above owes much to the members of a production system group at Carnegie-Mellon University. The members of this group, in addition to myself, are C. Forgy, J. Laird, P. Langley, A. Newell, and M. Rychener. I also want to acknowledge the helpful comments on the first draft of this paper from J. Bentley and D. Kosy.

REFERENCES

- [1] Forgy, C. A production system monitor for parallel computers. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1977.
- [2] Forgy, C. and J. McDermott. OPS, a domain independent production system language. *IJCAI*, 5, 1977.
- [3] McDermott, J. ANA: an assimilating and accommodating production system. Technical Report. Department of Computer Science, Carnegie-Mellon University, 1978.
- [4] McDermott, J. Some strengths of production system architectures. *NATO ASI on Structural/Process Theories of Complex Human Behavior*. Sijthoff, 1978.
- [5] McDermott, J. and C. Forgy. Production system conflict resolution strategies. In D. Waterman and F. Hayes-Roth (eds), *Pattern-Directed Inference Systems*. Academic Press, 1978.
- [6] Moore, J. and A. Newell. How Can Merlin Understand? In L. Gregg (ed), *Knowledge and Cognition*. Lawrence Erlbaum Associates, 1973.
- [7] Newell, A. Knowledge representation aspects of production systems. *IJCAI*, 5, 1977.
- [8] Rychener, M. Control requirements for the design of production system architectures. *Symposium on AI and Programming (SIGART/SIGPLAN)*, 1977.
- [9] Sussman, G. J. *A Computer Model of Skill Acquisition*. American Elsevier, 1975.