CMU-CS-78-156

ANA: AN ASSIMILATING AND ACCOMMODATING PRODUCTION SYSTEM

John McDermott

December, 1978

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.

This 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.

1. Introduction

Any AI system that hopes to ever 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 the knowledge that is has 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.

The system described, ANA, has a limited amount of knowledge about how to function in a simple environment. When presented with an unfamiliar task, it tries to determine whether it has knowledge relevant to that task. If it finds such knowledge and the knowledge enables it to perform the task, then it associates that knowledge with the task. If it encounters difficulties as it performs the task, it augments its knowledge in a way that enables it to avoid these difficulties when they subsequently arise. 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. Then as it uses its method, instead of executing the actions prescribed by the method, it executes the actions dictated by the mapping. Whenever ANA is able to use 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. ANA sometimes encounters difficulties that it cannot figure out how to remedy. When this occurs, ANA asks the person instructing it how to resolve the problem; it then builds a production that associates the instructor's remedy with the difficulty.

One of ANA's limitations should be noted from the start. ANA cannot "learn" new methods. The methods that it uses are all sets of hand-coded productions that form part of the initial contents of its production memory. The productions that it builds do in fact augment its abilities, but they do so by extending the domain of the methods that it already has. ANA could, of course, start off with a different set of methods or with additional methods, but it currently has no way of acquiring methods from scratch.

The body of the paper is divided into four sections. In the next section I describe in considerable detail the way in which ANA's knowledge is represented. In the third section, I

describe ANA's task environment (an automatic paint shop), discuss four of the tasks that ANA has been given, and display one of ANA's methods. The fourth section deals with assimilation; I discuss how ANA makes use of its methods to perform unfamiliar tasks by analogy and describe the various strategies that ANA uses when an analogy breaks down. The fifth section deals with accommodation; I describe the various productions that ANA builds in order to transform unfamiliar tasks and unfamiliar difficulties into familiar (and unproblematic) ones.

2. Some Context

One of my reasons for doing the research described below was to explore the degree of support that a production system architecture provides for assimilation and accommodation. Taking the architecture as given, I looked for a way of representing ANA's knowledge that would make it easy for ANA to transform unfamiliar tasks into familiar ones. After describing the representation that I developed, I will discuss the extent to which it constrains the choice of assimilation and accommodation strategies, and I will briefly characterize the strategy that ANA employs.[1]

2.1 How ANA Represents Its Knowledge

The production system architecture used to implement ANA is called OPS2 [Forgy and McDermott, 1977; McDermott, 1978; Newell, 1977]. 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 zero or more condition elements and zero or more 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. An instantiation is an ordered pair of a production and the elements from working memory that satisfy the conditions of the production. The production system interpreter operates within a control framework called the **recognize-act cycle**. In recognition, it finds the instantiations to be executed, and in action, executes one of them, performing whatever actions occur in the action side of the production. 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

[1] Anderson's production system, ACT, which models human cognitive processes, uses rather different assimilation and accommodation strategies [Anderson, Kline, and Beasley, 1978a and 1978b]. Comparing ACT's strategies with ANA's gives some hint of the space of strategies that are available to production systems. For a general discussion of the issues that face anyone designing a system with assimilation and accommodation capabilities, see Moore and Newell [1973].

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 Forgy [1977]). In conflict resolution, the interpreter selects (on the basis of a few simple rules) one instantiation to execute (see McDermott and Forgy [1978]). 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.

In order 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 of ANA's methods 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 type of the data element [see Rychener, 1977]. There are four types of data elements: **goals, constraints, percepts,** and **concepts.**

A data element marked "goal" contains two subelements (in addition to the marker). One of these subelements 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 (in addition to the marker); constraints are attribute-name-value triples. A goal element and the two sets of constraints that 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.

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

Goal description: Paint the yellow table in L32 red.

(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 2-1: A goal and a set of constraints

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.[1]

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 that is 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 is given that constrains the selection of an object and information about a particular object. Elements that contain the first sort of information are marked "constraint", while elements the

[1] @scan does not actually generate the token-name. Every object has a token attribute, and the value of this attribute (which can be thought of as ANA's private name for the object) is used to tie the bundle of percepts together. If @scan generated a token-name itself each time it was executed then ANA either would be unable to recognize two perceptual bundles as descriptions of the same object at different times or would need a fairly sophisticated recognition capability.

contain the second sort of information 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 that it refers to as table*1 is the one to be operated on.

Percept: There is a light weight, yellow table at the top of the stack in L32.

(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))

Concept: There is a table in L32 that satisfies the constraints on object*2.

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

Figure 2-2: A set of percepts and a concept

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 goal may be "not-achieved", "achieved", or may have the status "no-method". A constraint may be "given", "not-given", or "possible". Concepts and percepts may be either "true" or "false". The marker also contains information indicating the element's recency.

2.2 ANA's Assimilation and Accommodation Strategies

OPS2 (ie, the architecture itself) provides little direct support for assimilation. Assimilation involves mapping new knowledge into existing knowledge. The OPS2 interpreter, however, has a match algorithm with only the most primitive mapping capabilities. Each atom in a data element is a constant. Each atom in a condition element is either a constant (or a set of equivalent constants) or a variable. In order for a condition element to match a data element (ie, in order for OPS2 to generate a mapping), corresponding constants must be identical. Thus OPS2 can only map constants into identical constants or into a semantically impoverished variable. In order for a production system to be capable of assimilation, then, it must have knowledge that enables it to generate more interesting mappings.

OPS2 does, however, provide some support for accommodation. Once a production system has mapped new knowledge into existing knowledge and begins to perform some task, there are two types of difficulties that it can encounter. One type arises when the method that it is using is under-specified, ie, does not contain sufficient knowledge to enable the new task to be done in a satisfactory way. When a system finds that it is using an under-specified method, it must find a way of dealing with whatever problems it has created. If it is successful, its accommodation capability should provide it with a way of avoiding those problems in the future. OPS2 makes this relatively easy to achieve because of its conflict resolution rules. Each of the rules provides some support, but the principal support comes from the special-case rule. Given two instantiations, if the data of one is a proper subset of the data of the other, this rule selects the instantiation containing more elements. Thus the system can simply build a production that is a special case of the production which, if fired, would create a problem. The condition side of the new production can contain all of the condition elements of the faulty production plus one or more additional elements that are sensitive to the context within which the problem develops. The other type of difficulty arises when the method that is being used is over-specified, ie, contains expectations specific to a particular task or class of tasks, but not relevant in the current situation. If there is nothing in the unfamiliar situation that corresponds to these expectations, then the system does not know whether the expectations are relevant (and indicate by not being satisfied that it is in trouble) or irrelevant. Again, once the system finds out which, it wants to remember so that in future situations where they are not relevant they can simply be ignored. In order for OPS2 to provide direct support for this type of accommodation, it would have to provide a means of modifying existing productions. OPS2 provides only the most limited support for this.

Since OPS2 provides almost no direct support for assimilation, in order for a production system whose knowledge is organized as a collection of methods to assimilate new knowledge, it must know how to do two things. First, as I have said, it must know how to make contact between the knowledge given to it in the current task description and a relevant method. Second, it must know how to use the knowledge contained in the method to do the unfamiliar task. But these two requirements are somewhat at odds. Given the OPS2 match algorithm, the most likely approach to contact is to make the methods that the system has as general as possible. In particular, whenever an attribute is relevant but its particular value is of no concern, the value should be a variable. The problem with this approach is that it makes unfamiliar tasks more difficult to accomplish since little information is around if a method proves inadequate. If a system is using a method and runs into trouble, presumably the method is being used in a way not foreseen by its creator. In order for the system to figure out why the method is inadequate, it needs to know how the current situation differs

from the set of situations for which the method is adequate. But if the method is full of variables, that information is simply not available. Thus it will be highly dependent on whomever gave it its task for information about how to get out of the difficulty that it is in.

So that ANA will have some information to help it recover when one of its methods proves inadequate, each of its methods for operating on its environment is tailored to a specific situation. ANA has, for example, a method for carrying tables from one specific location in its environment to another specific location. Since this method is tailored to this one situation, it can be used effectively both for its intended class of tasks and for other quite different tasks. Since the class of tasks for which the method is intended is so constrained, it is likely that the method will be adequate for any task in the class. If the method is used to accomplish some task for which it was not intended, it will often turn out to be inadequate. But the knowledge of the differences between the task it was given and the task that the method knows how to accomplish will be easy to discover. ANA's response to an unfamiliar task is to map it into a task that one of its methods can accomplish, and then use that method analogically. In effect, ANA's strategy is to severely limit the immediate applicability of its methods so that when it is asked to perform an unfamiliar task, it will know exactly how to perform a comparable task, and it will know from the mappings all of the differences between that familiar task and the unfamiliar task. The expected gain is that this knowledge will enable it to modify its behavior in precisely the way needed to perform the unfamiliar task.

ANA's technique for dealing with method inadequacy is simply to watch for things to go wrong. It has some productions that are sensitive to the most common types of problems (eg, its inability to apply one of its operators to an object in the environment). Before it tries an unfamiliar task, it builds additional productions that are sensitive to specific problems that could arise (eg, its failure to realize some particular expectation). The reason for choosing this technique is one of efficiency. If a production system has a set of rules that are sensitive to manifestations of inadequacy, then the system does not need to continuously attend to the adequacy of its method; it can set up an initial set of mappings, assume that that set of mappings is adequate, and just cycle until some event occurs that forces it to recognize that a modified set of mappings is required. The alternative is to build the production system in such a way that it interrupts itself after each cycle, examines the consequences of the previous production firing for the task it is trying to accomplish, modifies its mappings if it sees that that is required, and then continues. The overhead cost of this alternative technique is very high, and it is not clear that anything would be gained by using it. ANA interrupts itself enough to keep itself on the right path most of the time. On those occasions when it does get off the path, the amount of time that it would take it to foresee that it was going astray may in fact be greater than the amount of time it takes it to correct its mistakes.

Though accommodation is simply the process by which the results of assimilation can be made permanent, the new knowledge can be stored in a variety of ways. After a system has successfully used a method to perform an unfamiliar task, it is in a position to make that method more general. It can try to modify the method in such a way that the method will now apply to the union of the class of tasks for which it was originally intended and the class of tasks which the task it just accomplished is an instance of. One possible approach to generalizing a method would be to find the constants in the description of the unfamiliar task that could be straightforwardly mapped into constants in the productions comprising the method; these constants could be replaced with variables. Then for each production containing a critical constant a second production could be built containing the corresponding constant from the description of the unfamiliar task. The advantages of this use of accommodation, if it works, are clear. But there are two difficulties. First, it is not clear how a system would decide if a mapping was "straight-forward"; however it would do it, surely it would frequently make mistakes. Thus it is possible that a system might learn more in the long run with an accommodation strategy that avoided generalization. The second difficulty has already been mentioned: the more general a method is, the more difficult it is to determine, in the context of a particular task, why it is inadequate. Thus if a method is generalized, it will be harder for a system to figure out what to do if the method breaks down than if it had not been generalized.

ANA's accommodation strategy is simply to preserve the results of its assimilation of new knowledge. Whenever it successfully accomplishes a task (or subtask) by analogy, it builds a production that associates the description of the new task with the method that it used to accomplish that task. It makes no modifications to the productions comprising this method. If ANA has to overcome difficulties in order to accomplish a task (ie, if the productions comprising the method are not by themselves adequate to accomplish the new task), ANA builds productions that are special cases of the original productions; these productions enable ANA to avoid the difficulties in the future. Since the original very specific method is always the method used, if the extended method proves inadequate, ANA has access to the information that will enable it to discover the differences between its current task and the task for which the method was intended.[1]

---

[1]Anyone familiar with HACKER [Sussman, 1975] 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.

3. ANA's Task Environment

ANA's task is to manage a paint shop. ANA operates a machine that sits in the middle of a shop. The only things in the shop other than the machine are a variety of paintable objects and some overhead lines that enable ANA to cart objects around. 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.

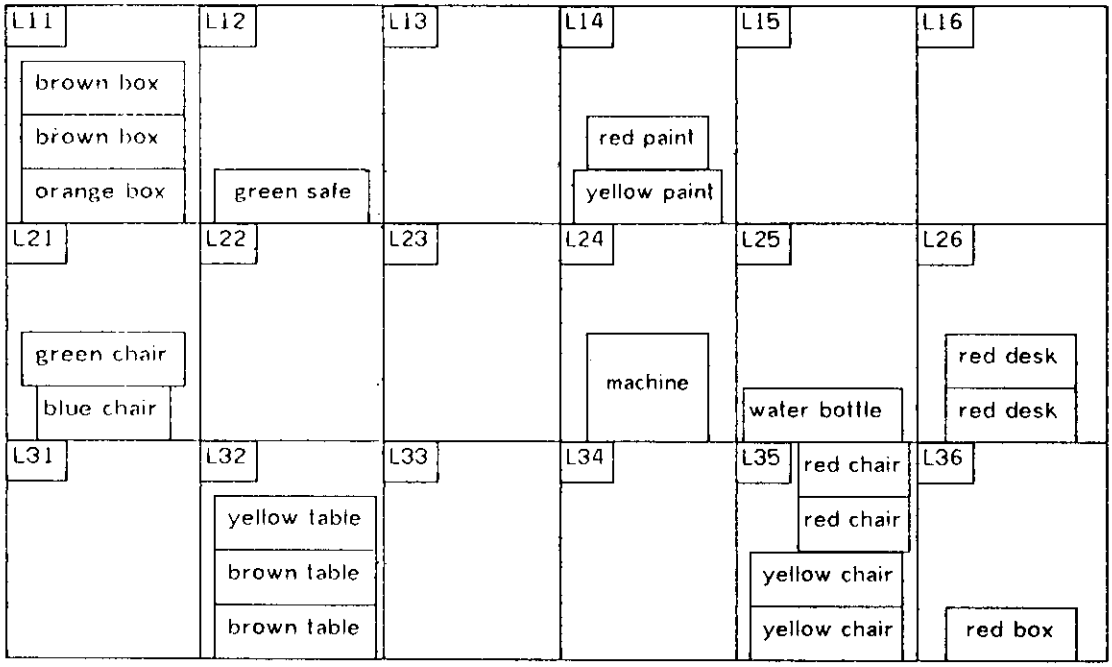

Figure 3-1: ANA's paint shop

ANA has five operations that it can perform on the objects in the shop. One of its operators, @spray, starts the machine. Three other 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 that object 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. It should be noted that in ANA's world some attributes are mutable and some are not. Weight is the only immutable attribute; color, state, position, and location are all mutable.

ANA's initial task-specific knowledge consists of six methods. ANA has a method for painting tables whose initial location is L32 red. ANA also has a method for clearing off the top of desks that are in L32. ANA's other four methods are all methods for transporting objects. One of these methods enables it to move boxes that are in L25; the location to which a box is to be moved is not specified. 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 known to be 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; this method is tailored for the situation in which the cabinet to be moved is really-heavy.

Four of the tasks that ANA has been given will be used as examples throughout the rest of the paper. The four tasks are:

- Task0: Paint the yellow table in L32 red and then move it to L16.
- Task1: Paint the blue chair in L21 red and then move it to L35.
- Task2: Wash the thing in L12.
- Task3: Paint the box at the bottom of the stack in L11 yellow.

Task0 is, for ANA, by far the most straight-forward of the four. To do this task, ANA must

@carry the table at the top of L32 to L23, then @carry the paint can at the top of L14 to L24, then @spray, then @carry the paint can at the top of L24 to L14, and finally @carry the table at the top of L34 to L16. Since ANA has a method for painting tables that are in L32 and a method for carrying tables from L32 to L23, much of this task can be done without using analogy; the subtasks that do require analogy are all easily mapped into ANA's method for carrying tables. Task1 is from ANA's point of view much more complex. When ANA tries to @carry the blue chair in L21 to L23, @carry will fail since the blue chair is not at the top of the stack; so ANA will have to clear off the top of the blue chair. When ANA carries the painted chair from L34 to L25, it will fall off of the stack since the stack already has four objects in it. Task2 creates other problems. When ANA tries to @carry the thing (the safe) in L12 to L23, @carry will fail since the safe is heavy; so ANA will have to select a different operator (@push). When ANA gets to the point in its method for painting tables where the submethod of carrying a paint can from L14 to L24 is generated, it must recognize that this submethod is inappropriate and instead carry the water bottle in L25 to L24. Task3 raises, among other problems, the problem of selecting among competing methods. ANA has a method for carrying cabinets that are not at the top of a stack and has a method for carting cabinets that are really-heavy. In this task, ANA must carry a box that is really-heavy and at the bottom of L11. Neither of the methods is adequate. For reasons that will be discussed below, ANA uses its method for carrying cabinets that are not at the top of the stack; @carry fails since the box is really-heavy; so ANA has to select a different operator (@cart). When ANA tries to @carry the yellow paint can to L24, @carry will fail since the paint can is not at the top of the stack; so ANA will have to clear off the yellow paint can.

In order to give an idea of what ANA's task-specific methods look like, its method for painting tables is shown in Figures 3-2, 3-3, and 3-4. ANA's paint method consists of six productions (paint1 - paint6).[1]. The conditional part of paint1 consists of seven condition elements. The first of these, the element marked "goal", contains two variables that associate a set of action constraints with a set of object constraints. The next three conditions are constraints on the action: it must be of type paint, the effect of the action must be to change the surface of an object, and the expected change to the surface is that it will become red. The final three condition elements are constraints on the object to be acted on: it must be of type table, it must occupy a location, and the label of that location must be L32. The action part of the production consists of five elements that describe a subgoal that must be

[1] A literal atom whose name begins with "=" is a variable. A variable may match any single data subelement; if a variable occurs more than once in a production, all occurrences must match equal subelements. The operator, <bind>, returns a symbol different from the symbols returned on previous calls to <bind>. If <bind> is given a variable name as its argument, the value returned by <bind> is bound to that variable

IF the goal is to paint a table in L32 red, THEN generate the subgoal of finding an object in L32 that satisfies the constraints on the object to be acted on.

```
paint1  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (color =effect (constraint given =time-a) red)
          (type =object (constraint given =time-o) table)
          (location =object (constraint given =time-o) =location)
          (label =location (constraint given =time-o) L32)
         -->
          ((<bind> =new-act) =object (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) find)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) mind)
          (location =new-effect (constraint given =new-time) (<bind> =location-effect))
          (label =location-effect (constraint given =new-time) L32))
```

IF the goal is to paint a table in L32 red and there is an object in L32 satisfying the constraints on the object to be acted on, THEN generate the subgoal of carrying that object to L23.

```
paint2  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (color =effect (constraint given =time-a) red)
          (type =object (constraint given =time-o) table)
          (location =object (constraint given =time-o) =location)
          (label =location (constraint given =time-o) L32)
          (token =object (concept true =) =token1)
          (token =location (concept true =) =token2)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L32)
         -->
          ((<bind> =new-act) =object (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) carry)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) location)
          (location =new-effect (constraint given =new-time) (<bind> =location-effect))
          (label =location-effect (constraint given =new-time) L23))
```

Figure 3-2: ANA's paint method

```
paint3  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (color =effect (constraint given =time-a) red)
          (type =object (constraint given =time-o) table)
          (token =object (concept true =) =token1)
          (token =location (concept true =) =token2)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L23)
        -->
          ((<bind> =new-act) (<bind> =new-object) (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) carry)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) location)
          (location =new-effect (constraint given =new-time) (<bind> =location-effect))
          (label =location-effect (constraint given =new-time) L24)
          (type =new-object (constraint given =new-time) pntcan)
          (location =new-object (constraint given =new-time) (<bind> =new-location))
          (label =new-location (constraint given =new-time) L14)
          (label =new-object (constraint given =new-time) red))


paint4  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (color =effect (constraint given =time-a) red)
          (type =object (constraint given =time-o) table)
          (token =object (concept true =) =token1)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L23)
          (type =other-object (constraint given =time-oo) pntcan)
          (label =other-object (constraint given =time-oo) red)
          (token =other-object (concept true =) =token3)
          (location =token3 (concept true >time) =token4)
          (label =token4 (concept true >time) L24)
        -->
          (@spray)
          ((<bind> =new-act) =object (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) find)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) mind)
          (color =new-effect (constraint given =new-time) red)
          (location =new-effect (constraint given =new-time) (<bind> =location-effect))
          (label =location-effect (constraint given =new-time) L34))
```

Figure 3-3: ANA's paint method, continued

```
paint5  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (color =effect (constraint given =time-a) red)
          (type =object (constraint given =time-o) table)
          (token =object (concept true =) =token1)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L34)
          (color =token1 (concept true >time) red)
          (type =other-object (constraint given =time-oo) pntcan)
          (label =other-object (constraint given =time-oo) red)
          (location =other-object (constraint given =time-oo) =location)
          (token =other-object (concept true =) =token3)
          (location =token3 (concept true >time) =token4)
          (label =token4 (concept true >time) L24)
         -->
          ((<bind> =new-act) =other-object (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) carry)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) location)
          (location =new-effect (constraint given =new-time) (<bind> =location-effect))
          (label =location-effect (constraint given =new-time) L14)
          (label =location (constraint given =time-oo) L24))

paint6  ( (=act =object (goal not-achieved =time)) & =w
          (type =act (constraint given =time-a) paint)
          (effect =act (constraint given =time-a) =effect surface)
          (type =object (constraint given =time-o) table)
          (token =object (concept true =) =token1)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L34)
          (color =token1 (concept true >time) red)
          (type =other-object (constraint given =time-oo) pntcan)
          (label =other-object (constraint given =time-oo) red)
          (token =other-object (concept true =) =token3)
          (location =token3 (concept true >time) =token4)
          (label =token4 (concept true >time) L14)
         -->
          (<delete> =w) (=act =object (goal achieved =time)) )
```

Figure 3-4: ANA's paint method, continued

achieved in order for the painting task to be accomplished: the subgoal is to look in the shop for an object satisfying the object constraints given in the conditional part of the production and notice the label of that location. The second production, paint2, has a conditional part consisting of 11 condition elements. The first seven are identical to the seven condition elements in paint1. The eighth condition element matches a data element that links the description of the object to be acted on with the token-name of an object fitting that description. The ninth matches a data element that links the description of the location of the object to be acted on with the token-name of a location fitting that description. The final two condition elements stipulate values that the two token-names must have. Put another way, if ANA is given the goal of painting the table in L32 red, and if it has found a table in L32, then paint2 can fire. The result of firing paint2 is that the subgoal of carrying the table in L32 to L23 is generated. The other four productions comprising ANA's paint method have the same form as paint1 and paint2. Paint3 can fire when the table to be painted is known to be in L23; the result of firing paint3 is that the subgoal of carrying the paint can labeled red from L14 to L24 is generated. Paint4 activates the paint machine and generates the goal of checking to make sure that a red object was output. Paint5 generates the subgoal of carrying the paint can back to L14. Paint6 can fire when ANA has accomplished its task.

4. Assimilation

To do the tasks described in the previous section, ANA has to be able to assimilate new knowledge. Since the few methods that ANA 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 that it does have to the unfamiliar tasks. ANA's strategy is to find a method that is adequate for a related task and assume that this method will be adequate for the new task. Since this assumption will inevitably (and 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. If it 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 tries the task again keeping those constraints in mind. If it sees that it is unable to achieve a result because of some constraint imposed by the method it is using, it checks with its instructor to see if that constraint is necessary; if the instructor indicates that it is not necessary, ANA ignores the constraint and continues with what it is doing. In this section, I will first describe how ANA makes contact with an appropriate method and how it maps the task that this method can accomplish into the task it is given. Then I will describe how ANA recovers when a method turns out to be inadequate for the task that is given.

4.1 Contact and Mapping

When ANA is given the task of painting the yellow table in L32 red (task0), it recognizes the task as one that it knows how to accomplish. Since it has a method for painting tables in L32 red, it simply uses that method. When its paint method generates the subgoal of carrying the table from L32 to L23, it uses its method for carrying tables to accomplish this subtask. But when the painting method generates the subgoal of carrying red paint from L14 to L24, ANA is faced with an unfamiliar task. At this point ANA has to make contact with one of its existing methods. An obvious candidate, of course, is its method for carrying tables from L32 to L23, but how is ANA to know this? It needs some way of making contact with the method. In order to enable ANA to use one of its methods to accomplish an unfamiliar task, each method has associated with it a **method description production.** These method description productions enable ANA to relate unfamiliar tasks to familiar ones.

The method description production for ANA's paint method is shown in Figure 4-1. The conditional part of this production differs from the conditional part of the productions that comprise the actual method (shown in Figures 3-2, 3-3, and 3-4) in four ways: (1) The initial condition element is marked "goal no-method" rather than "goal not-achieved". (2) The condition elements specifying the action type and object type are marked "constraint not-given" rather than "constraint given". (3) Each of the other constraints is marked "constraint given", but has no value specified for its attribute. (4) The final two condition elements have nothing corresponding to them in the productions that actually comprise the method. The "goal no-method" marker in the first condition element provides a way of insuring that no method description production will fire unless ANA has no method for the task it is given. The two condition elements marked "constraint not-given" provide ANA with a way of making contact between its existing methods and the task at hand. The purpose of the other constraints is simply to test for the availability of information that might have some bearing on how the task is done. Of the final two conditions, the negated one[1] insures that ANA will not consider the same method twice as it examines alternative methods; the other condition element guarantees that the method description production will fire as soon as it is enabled. The action part of the production contains the set of constraints that the paint 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.

---

[1] A "-" in front of a condition element indicates that the production containing the element can be enabled only if there is no element in working memory that matches it.

```
paint*  ( (=act =object (goal no-method =))
          (type =act (constraint not-given =) paint)
          (effect =act (constraint given =) =effect surface)
          (type =object (constraint not-given =) table)
          (location =object (constraint given =) =location)
          -(analog (paint table) (concept true =) (=act =object =effect) 2)
          (cycle ! =)
        -->
          (analog (paint table) (concept true (<bind> =new-time)) (=act =object =effect) 2)
          (type =act (constraint possible =new-time) paint)
          (effect =act (constraint possible =new-time) =effect surface)
          (color =effect (constraint possible =new-time) red)
          (type =object (constraint possible =new-time) table)
          (location =object (constraint possible =new-time) =location)
          (label =location (constraint possible =new-time) L32) )
```

Figure 4-1: A Method Description Production

4.1.1 Method Selection

In order for ANA to be able to make contact between its methods and unfamiliar tasks, it needs some way of determining what actions are related to the action that is specified by the task and what objects are related to the object that is to be acted on. Part of its production memory contains such knowledge. The knowledge is represented as a tree of types; each node of the tree is defined by a set of productions. Each of these productions has as one of its condition elements the node's type; this element is marked "constraint not-given". One of the productions has as one of its action elements an element marked "constraint not-given" indicating the supertype of that type; each of the other productions has as one of its action elements an element marked "constraint not-given" indicating an instance of that type. Another condition in each of the productions encodes information about direction (up or down) so that from any node, ANA can either go up to a supertype or down to an instance. Initially, ANA knows the type relationships of the 8 actions and the 15 things shown in Figure 4-2.

When ANA is given an unfamiliar task, it generates the goal of searching its semantic net of action types and its semantic net of object types for actions and objects related to those given in the description of its task. It first asserts the type of the stipulated action and the type of the object to be acted on; both assertions are marked "constraint not-given". If ANA has productions with condition elements matching these two types, then all of the downward

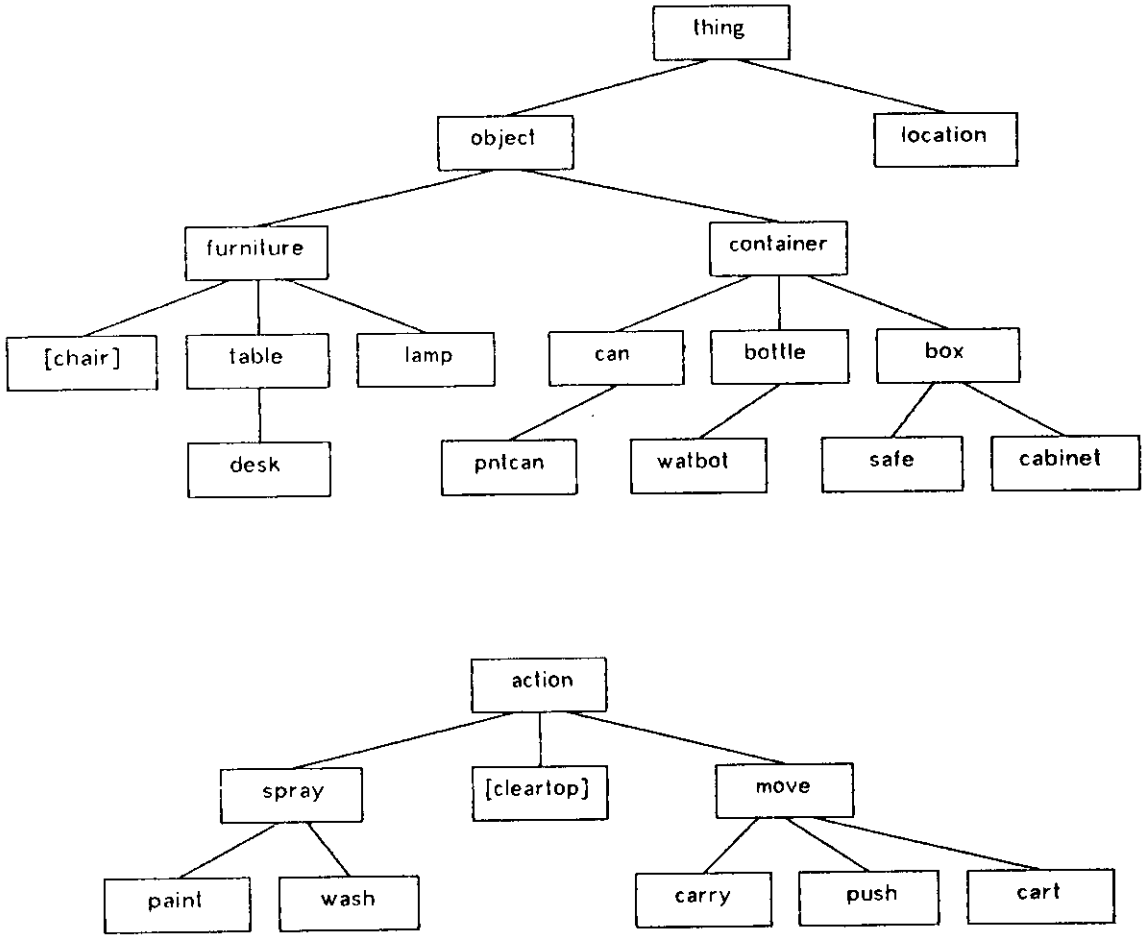

Figure 4-2: An action net and an object net

arcs from the two nodes are traversed (alternatingly) in a depth first search.[1] When all of the downward nodes have been touched, ANA goes up one level and then traverses all downward arcs that it has not yet traversed. Each time a production fires, the possibility exists that one of ANA's method description productions has become satisfied. If one does, it fires, and the method it describes is evaluated; then if the evaluation suggests that continuing the search for alternative methods is warranted, control is returned to the net and the process of traversing the net continues. Since the two nets that ANA must explore are so small, the

[1] If ANA does not have a production for one of the types, it asks the instructor for that type's supertype. When the instructor responds, ANA builds a pair of productions. One of them will assert the supertype whenever ANA is searching upward and an element marked "constraint not-given" and containing the type is asserted. The other will assert the type whenever ANA is searching downward and an element marked "constraint not-given" and containing the supertype is asserted. ANA adds the nodes for "chair" and "cleartop" (see Figure 4-2) in the course of doing the four tasks.

depth first search strategy that it employs does not consume a great deal of time. It is probably the case that for larger (ie, branchier) nets, a more intelligent search strategy would be required.

If ANA is given the task of washing the safe in L12 (Task2), its semantic net mechanism will, after a few cycles, enable the method description production for painting tables, and that production will fire. At that point, ANA will evaluate its method for painting tables. 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 first 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 painting tables. 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. Thus, given the task of washing the safe, ANA would assert that the action type "paint" should be mapped into the action type "wash", the description of the expected effect of painting should be mapped into the expected effect of washing, the object type "table" should be mapped into the object type "safe", and the description of the location of the table, that it is in L32, should be mapped into the description of the location of the safe, that it is in L12. If the description of the pseudo-object includes attributes that are not contained in the description of the actual object, ANA will look at the actual object. For each new attribute it finds, ANA will assert an element that indicates that the pseudo-value is to be mapped into the perceived value. If the description of the pseudo-object contains attributes that the actual object does not have, ANA will assert an element that indicates that the pseudo-value is to be mapped into itself. If the description of the expected effect asserted by the method description production includes an attribute that is not included in the description of the expected effect of the actual action and vice versa, ANA will assert an element that indicates that the unmapped pseudo-attribute-value pair is to be mapped into the unmapped stipulated attribute-value pair. After having established these mappings, the pseudo-value of each of the attributes asserted by the method description production will have been mapped into some stipulated (or perceived) value. There may be attributes in the actual description whose stipulated values are not a part of any of these mappings, but this is all right since the method that is to be used analogously could not make use of them because all of the attributes that it is equipped to deal with are asserted by the method description production.

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

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 ANA associates a pre-condition with an action type, it also notes whether it can do anything to satisfy that pre-condition if it is not satisfied. If, for example, the pre-condition is that in order to carry an object, the object must have nothing on top of it, ANA notes that the position of an object in a stack is mutable. If the pre-condition is that in order to carry an object, the object must be light, ANA notes that the weight of an object is immutable. If the pre-condition is that in order to carry an object, the location in which the object will be put must contain fewer than four objects, ANA notes that the composition of a location is mutable. When ANA's evaluation of a method begins, if it has any productions that contain information about the pre-conditions on the action type of the method being evaluated, those productions will fire. ANA checks to determine whether any of the pre-conditions are violated. If a pre-condition is violated by an immutable attribute of an object, ANA will conclude that the method it is considering is unlikely to work. If there are no immutable violators, but a pre-condition is violated by mutable attribute, ANA will conclude that the method will probably work, but that it may have a better method. If no pre-conditions are violated, ANA will conclude that the method will probably work.

After ANA has evaluated a method, it typically returns to its semantic net and generates additional types with the hope that a better method will turn up. How long it spends in searching for other candidates depends on how good its current best candidate is; it spends more time if the current best candidate falls into the "unlikely to work" category than if it falls into the "will probably work" category. When another candidate proposes itself, ANA evaluates it and then compares it to the current best previous candidate. If one of the candidates is in the "unlikely to work" category and the other is not, the unlikely to work candidate is rejected. If both are in the same category, ANA checks to determine whether one of the methods is a "special case" of the other; if the pseudo-goal-description asserted by one of the methods contains more constraints than the pseudo-goal-description asserted by the other, the method with more constraints is deemed a special case and is preferred. If neither method is a special case of the other, ANA checks to determine if more identity mappings are associated with one of the methods than with the other; ANA assumes that the more similar the pseudo-values and the stipulated values, the more likely the method is to

work, and so prefers that method with the largest number of identity mappings. If the two methods have the same number of identity mappings, ANA rejects the more recent candidate on the assumption that since it is "farther" from the actual task in the semantic net, it is less likely to work.

To see how this evaluation process works, consider the case in which ANA is given the task of carrying the really-heavy box at the bottom of the stack of boxes in L11 to L23 (Task3). ANA has four methods that it could consider. For ease of reference, I will refer to each by its action and object type; the methods, then, are "carry table", "move box", "carry cabinet", and "cart cabinet". As explained above, the method description productions for each of these methods have a different set of condition elements; the set that each has specifies the set of attributes that are of concern to its method. In addition to the attributes that all method description productions have (action type, object type, and effect), the "carry table" method is sensitive to the location of the object to be carried and to the location to which it is to be carried. The "move box" method is sensitive only to the location of the object to be moved. The "carry cabinet" method is sensitive to the location of the object to be carried, the location to which it is to be carried, and the position of the object in the stack. The "cart cabinet" method is sensitive to the location of the object to be carted, the location to which it is to be carted, and the weight of the object. If ANA considered all four of these methods before it knew of any pre-conditions on carry, move, and cart, the methods would all be put in the category of "will probably work". In comparing the methods, ANA would prefer "carry cabinet" and "cart cabinet" to the other two methods since both are special cases of the other two. ANA would prefer "carry cabinet" to "cart cabinet" since that method has an identity mapping (the action type "carry" in the method description is mapped into the action type "carry" in the description of the task). If ANA considered the methods after learning that to carry an object the object must be light, its evaluation would be somewhat different. The "carry table" and "carry cabinet" methods would be put in the category of "unlikely to work". The "move box" and "cart cabinet" methods would be put in the category of "will probably work". Thus the latter two methods would be given preference. The "cart cabinet" method would be selected in preference to the "move box" method since it is a special case.

4.1.2 Using a Method Analogically

After 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 resulted only in a set of constraints marked "constraint possible", ANA still cannot do anything. The solution is, however, straight-forward. ANA simply 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 given 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 of the information that it needs in order to distinguish between what it is actually trying to do ("constraint given"s with no corresponding "constraint possible"s) and what it is pretending to do ("constraint given"s with corresponding "constraint possible"s). 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); so 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.

The only unmentioned ingredient necessary to make this work is to provide ANA with a way of noticing when it is running into difficulties. A number of different types of difficulties can arise. One type of difficulty is the "unexpected result" difficulty. As you may remember, ANA's method description productions typically include one or more condition elements that indicate the expected effect of the method. As ANA prepares to make use of a method it has selected, 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. Thus ANA builds a production that keeps its eye, so to speak, on the object; the production is specific to the particular task being attempted, and thus once the goal is achieved, the production can no longer fire. If after the object is painted it is some color other than yellow, this production can fire. If it fires, the person giving the task is asked for some information that will enable ANA to keep the unexpected result from occurring on subsequent occasions. (What knowledge is requested and how it is used is discussed below.) It is sometimes the case, of course, that a goal cannot be achieved by a single action. Whenever ANA attempts to use its method for clearing off the top of a desk (since it expects that the result of moving an object that is on top of the thing to be cleared off is that the thing to be cleared off will be at the top) it builds a production that looks for this result. However, if it is attempting to clear off an object that has more than one object on top of it, moving just one object will not produce the expected

result. ANA is sensitive to this possibility, and so when it builds the production, it arranges things so that it will fire only if there are no productions in the method it is using that might enable the expected result to be achieved.

Once ANA has built the productions that watch for unexpected results, it is ready to make use of the method it has selected. ANA replaces the element marked "goal no-method" with an identical element marked "goal notachieved". At this point (presumably), one of the productions in the selected method will fire. In the rest of this section, we will consider two things that ANA must do in order to maintain the integrity of its mappings. The first of these has to do with the elements marked "concept true" that are in the condition side of some of ANA's productions; these elements specify the expected values of relevant attributes of the objects in the environment. When ANA uses a method analogously, these values, like the values of the constraints generated by the method description production, are likely to be different from the perceived values. Thus ANA must have a way of pretending that the objects currently of interest have these pseudo-values. Secondly, most of ANA's methods presuppose submethods. ANA's "paint table" method, for example, generates the goal of carrying the table; ANA's "carry cabinet" method generates the goal of clearing off the top of the cabinet; and ANA's "cleartop desk" method generates the goal of finding a desk. Thus, in order for ANA to be able to use a method analogously, it must be able to extend its initial set of mappings across a number of methods.

Before I say anything more about the elements marked "concept true", I had better say a few things about ANA's knowledge of the objects in the paint shop environment. Currently, all of ANA's knowledge about particular objects comes from looking at those objects. ANA has only one method that it can use when it wants to learn (or verify) something about an object in its environment. This method, whose action type is "find", is the only method that ANA has that can assert elements marked "concept true". Like the other productions that ANA has for operating on the objects in its environment, the conditional part of the "find" productions contains constraints on the object that is to be looked for and constraints that indicate the expected effect of finding the object. Here, however, the expected effect is not some change in the object, but a change to ANA (a change in ANA's awareness). When ANA is given the goal of finding some object, it first checks to determine if the description of the object to be found constrains the object's location; if so, it finds (ie, looks at) the specified location. Then it collects all of the other constraints on the object, with the exception of a type constraint if one is given, and @scans for an object satisfying that description. If an object satisfying the description is found, the set of percepts comprising that object are deposited in working memory. ANA then checks to determine if the type of the object scanned is consistent with the type stipulated in the description of the actual object desired. If the value of the percept whose attribute is type is the same as the value of the constraint

whose attribute is type, ANA knows that it has found the object it wants; if not, ANA uses its semantic net to determine if the value of the constraint is a supertype of the value of the percept. More concretely, if there is a red desk in location L36, and ANA is told to find the red thing in location L36, it will first look at (@scan for) location L36; it will then look at the red object in that location; and finally, it will check to determine is a desk is an instance of a thing. After ANA has found the desired object, it attends to the expected effect attributes; for each, it generates an element marked "concept true" that is identical to the element marked "percept true". The "find" method differs in a significant way from the other methods that ANA has for operating on its environment since the "find" productions are aware of the "constraint given"/"constraint possible" distinction. If this were not the case, of course, when "find" was invoked, ANA would be unable to distinguish between the actual object desired and the pseudo-object specified by the method currently being used. In addition to its principal function as a finder of objects, the "find" method also remarks any element marked "concept true" as "concept false" whenever its value is different from a more recently asserted percept (with the same attribute and token-name).

Knowing this much about "find", we can now return to the question of how ANA insures that productions that include condition elements marked "concept true" will be satisfied when a method is being used analogously. ANA has a production whose sole purpose is to watch for elements marked "concept true" to be asserted. Whenever this happens, if there is an element in working memory that maps a pseudo-value into a stipulated value and if the element just asserted has that stipulated value, the production fires; it asserts an element marked "concept true" whose value is the pseudo-value. A complementary production watches for elements marked "concept false" to be asserted. Again, if there is an element in working memory that maps a pseudo-value into a stipulated value and if the element just asserted has that stipulated value, the production fires; it asserts an element marked "concept false" whose value is the pseudo-value. It should be noted at this point that if the "find" method really marked as "concept false" all of the elements whose value was different from a more recently asserted percept, the work of the first of these productions would be for nought (and the second of these productions would be unnecessary). What the "find" method actually does is mark as "concept false" all of the elements falsified by some percept except those that have the pseudo-value of some map element.

The issue of how mappings are to be extended across methods is somewhat more complex than the issue of how elements marked "concept true" are to be included in the mapping. There are really two issues corresponding to the two sorts of subgoals that can be generated by a method. Whenever a subgoal is generated a new (distinct) description of the action and its expected effect is asserted. But the description of the object that is to be acted on may or may not be new. The production named paint2 (see Figure 3-2) generates

the subgoal of carrying a table. The table to be carried is the one that is to be painted, and no additional constraints on the table are asserted; the expected effect of the carrying is that the table will end up in location L23. Now assume that ANA is given the task of painting the box in location L11 yellow (Task3). Since the "paint table" method will be used analogically, ANA, before any of the "paint table" productions fire, will have mapped table into box, location L32 into location L11, and red into yellow. When the goal of carrying the table is generated, ANA check to see if the mappings can be extended. It 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. In this case, since none of ANA's map elements contain L23 as a pseudo-value, ANA simply assumes that carrying the object to L23 is the appropriate thing to do. A more intelligent system would, of course, try to figure out at this point whether there is enough of a difference between painting boxes and painting tables to suggest that the box to be painted should be carried to some location other than L23. ANA does not do this, but for the simple tasks it has attempted, this has not caused problems. The case with paint1 is a little different. This production, like paint2, does not introduce a new object; it generates the goal of finding the object to be painted. The object to be found has already been described (and since, as mentioned above, "find" is aware of the "constraint given"/"constraint possible" distinction, the object found will be the box in location L11). But unless ANA modifies the description of the expected effect of finding the object, it will be inconsistent with the description of the actual object. Thus in this case the initial mapping is extended to include this new description of an expected effect. Using the map element whose pseudo-value is L32, ANA asserts two label constraints: one is marked "constraint possible" and its value is L32; the other is marked "constraint given" and its value is L11.

Paint3 is an example of a production that generates a goal with a new object description as well as a new action description. The action type of the goal is "carry"; the object to be carried is the paint can labeled red in location L14; and the expected effect of the action is that the paint can will end up in location L24. Whenever ANA is doing a task by analogy and generates a goal containing a new object description, it tries to figure out, on the basis of the map elements it has asserted together with the little knowledge that it has about painting and such, whether it should modify the description of the object. ANA has a production that watches for the assertion of subgoals containing the description of a new object. If ANA's task is to paint the orange box in location L11 yellow (Task3), when paint3 fires, that production will then fire and generate the goal of determining whether a new object

description should be substituted for the one asserted. ANA checks to see if it knows anything about the purpose of this new object -- in this case, anything about the purpose of a paint can. As it happens, 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 yellow, ANA assumes that what it should carry is a yellow paint can rather than a red paint can. Since ANA's reasoning often involves some questionable assumptions, before it tries to achieve a subgoal containing the description of a new object, it asks whomever is giving it instructions whether it should assert its substitute object description or should assert an object description to be provided by the instructor.

If ANA is given the task of washing the thing in location L12 (Task2), it will, when paint3 fires, reason in a similar fashion. After recalling that the purpose of a paint can is to contain paint and that paint is for painting, it will notice that what it actually wants to do is wash. So it will try to find an object that has the same relationship to washing as a paint can has to painting. It finds that water is what is used for washing and that water bottles contain water. It assumes that it should substitute something of type water bottle for the paint can. It looks around the paint shop for a water bottle and when it finds one, assumes that that water bottle 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. Then, as before, it checks with the instructor before asserting this substitute description.

The two sets of productions just described, the set that insures that whenever appropriate if a concept containing a perceived value is asserted, a concept containing the corresponding pseudo-value will be asserted, and the set that insures that descriptions of objects and of expected effects will be consistent across methods, are sufficient to enable ANA to engage in the tasks described above without becoming confused. Of course, as ANA attempts to use its methods analogically, it frequently runs into difficulties since the methods are not perfectly suited to their new uses. In the next section we will consider how ANA responds to these difficulties.

4.2 Error Recovery

ANA can run into three kinds of trouble as it attempts to use its methods to perform new tasks. We have already seen one of them: when ANA is using a method analogously and generates a subgoal, it may not be obvious how to extend the mapping. If the subgoal contains the description of a new object (as well as the description of a new action), ANA attempts to discover the analogical relationship between that object and the actual task it is to perform. If ANA decides that the object description is inappropriate, it substitutes (with the instructor's approval) a different object description. If the subgoal contains only the description of a new action, ANA assumes that the action will have the appropriate effect. The other two kinds of trouble that ANA can run into are less intimately linked with analogy; the problems can arise whether or not ANA is using a method analogously -- though they arise more frequently and in a more pronounced form within an analogical context. One of these problems is that ANA can use a method that is **under-specified**. That is, the description of the object to be acted upon is insufficiently constrained, and so ANA ends up operating on (or trying to operate on) some object in an inappropriate way. The other trouble, almost the complement of the previous one, is that ANA can use a method that is **over-specified**. That is, the description of the expected effects of an action is overly constrained, and so ANA achieves a goal, but does not know it.

ANA has two ways of determining that it is using an under-specified method (and is in trouble). 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 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. If @carry fails because the object to be carried is heavy, the "action failed" element will indicate that the offending attribute is "weight" and that @carry will work only if the object is light. 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. ANA knows, for example, that weight is an immutable attribute and that position and location are mutable. Since ANA responds differently depending on whether the offending attribute is mutable or immutable, we will consider each case separately.

When ANA is given the task of painting the blue chair red (Task1), it first finds the blue chair and then attempts to carry it. Since the blue chair has another chair on top of it,

@carry fails, and an element containing the information that @carry failed because the blue chair is not at the top of the stack is deposited in working memory. The production watching for such an element fires, and the goal of recovering from this error is generated. When ANA checks to determine whether the position attribute is mutable or immutable, it finds that it is mutable. ANA's way of recovering from errors caused by mutable attributes is quite straight-forward. It simply builds a production that will fire whenever it considers using analogously a method whose action type is carry. This production asserts that in order for a carry method to work, the object to be carried must be at the top of a stack. Having built this production, ANA reconsiders what method to use in order to carry the chair. As we have seen, when ANA evaluates the appropriateness of a method, the first thing it checks is whether there is any reason to think that the method will not work. Since the production that it built will fire as soon as it begins to reconsider what method to use, ANA will know that in order to use a "carry" method, it will first have achieve a subgoal that will put the chair in the right position. ANA has some knowledge about how to satisfy pre-conditions. It knows, for example, that if an object has to be at the top of a stack and is not, that it should generate a cleartop goal. ANA does this; the blue chair becomes the top chair, and ANA is then able to use its carry method to achieve its goal.

When ANA is given the task of washing the thing in L12 (Task2), it first finds the thing and then attempts to carry it. This time @carry fails because the thing it is supposed to carry (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 selects one of its operators that has the same effect as @carry and tries it. Assume that it selects @push. It builds a production that masks (ie, that is selected in preference to) the production whose action side contains @carry. (I will discuss in some detail in the next section how this production gets built.) The function of the production is to determine whether the object to be carried is light, and it accomplishes this simply by scanning for the object. 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 production contains the operator @push. If ANA is trying to move the thing in L12, the first of the pair of productions built will fire and this will result in ANA looking at the thing to be moved. Since the weight of the thing is "heavy" (ie, not "light"), the second of the productions will fire; this time, ANA's attempt to move the safe is successful.

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, before actually beginning to

use it, 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 the production watching for the result of that operation, the production can fire. If 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".

Figure 4-3 provides an example of a production that ANA built to watch for an unexpected result. The production was built while ANA was doing Task1. Part of Task1 involves carrying the chair, after it is painted, from L34 to L35. The production shown 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 teeters and the chair ends up in location L34. When the production that watches for this unexpected result fires, the instructor 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 instructor indicates that nothing need be done. Then the instructor is asked to indicate what went wrong. All he need say is that the there are too many objects in L35; there have to be fewer than four. ANA then builds a production that contains the information that in order to be able to carry something somewhere, the location that the object is being carried to has to contain fewer than four objects. After building the production, it reconsiders what method to use for the task. This time around, when it considers carrying the chair, it realizes that in order to achieve its goal it first has to achieve the subgoal of moving one of the objects in L35 somewhere else.

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 watches 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 instructor whether the value of the attribute has to be the stipulated value. If the instructor answers "yes", then ANA keeps looking until it finds an object with the stipulated value. If the instructor answers "no", ANA assumes that the method it is using is over-specified, and asserts an element, marked "concept true", that indicates that the object that it has found has the stipulated value (even though it does not). Figure 4-4 shows two of the productions that are part of ANA's "move box" method. If ANA is given the task of moving the green chair, move2 will fire and generate the subgoal of finding an empty location (to move the chair to). Notice that ANA's "move box" method expects the label of the

```
p1-52   ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) carry)
          (effect =act (constraint given =time-a) =effect location)
          -(effect =act (constraint possible =time-a) =effect location)
          (location =effect (constraint given =time-a) =location)
          (label =location (constraint given =) L35)
          (location object (action done =op-time & >time) (chair*2 =token1) =quoted-op)
          (location chair*2 (concept true >op-time) =token2)
          -(location chair*2 (concept true >op-time) =token1)
         -->
          (<write> my attempt to carry chair*2 failed)
          (<write> chair*2 has location =token2 rather than =token1)
          (<write> how can i undo the effects of carry ?)
          (<read>)
          (<write> where did i go wrong ?)
          (create refinement (goal not-achieved (<bind>))
                      =time (=act =object =location) carry (<read>) =quoted-op))
```

Figure 4-3: A production that watches for an unexpected result

```
move2  ( (=act =object (goal not-achieved =time))
         (type =act (constraint given =time-a) move)
         (effect =act (constraint given =time-a) =effect location)
         (type =object (constraint given =time-o) box)
         (location =object (constraint given =time-o) =location)
         (label =location (constraint given =time-o) L25)
         (token =object (concept true =) =token1)
         (token =location (concept true =) =token2)
         (location =token1 (concept true >time) =token2)
         (label =token2 (concept true >time) L25)
        -->
         ((<bind> =new-act) (<bind> =new-object) (goal not-achieved (<bind>)))
         (type =new-act (constraint given (<bind> =new-time-a)) find)
         (effect =new-act (constraint given =new-time-a) (<bind> =new-effect) mind)
         (composition =new-effect (constraint given =new-time-a) empty)
         (label =new-effect (constraint given =new-time-a) L33)
         (type =new-object (constraint given =new-time-a) location)
         (composition =new-object (constraint given =new-time-a) empty))


move3  ( (=act =object (goal not-achieved =time))
         (type =act (constraint given =time-a) move)
         (effect =act (constraint given =time-a) =effect location)
         (type =object (constraint given =time-o) box)
         (location =object (constraint given =time-o) =location)
         (label =location (constraint given =time-o) L25)
         (token =object (concept true =) =token1)
         (token =location (concept true =) =token2)
         (location =token1 (concept true >time) =token2)
         (label =token2 (concept true >time) L25)
         (composition =other-object (constraint given =time-oo) empty)
         (token =other-object (concept true =) =token3)
         (composition =token3 (concept true >time) empty)
         (label =token3 (concept true >time) L33)
        -->
         (@carry =token1 =token3) )
```

Figure 4-4: Two of ANA's "move box" productions

empty location that it finds to be L33. But suppose that the first empty location that ANA finds is L22. Since the stipulated value of the label constraint is L33, while the value of the percept is L22, ANA's production that watches for the possibility of over-specified methods will fire. If it is the case that for the current task there is no reason for moving the chair to L33 rather than to L22, the instructor, when asked, will indicate that the value does not have to be L33. But move3 will not fire unless there is an element, marked "concept true" in working memory that indicates that the value of the label of the location that the object is to be moved to is L33. So ANA pretends that this counter-intuitive situation obtains, and asserts a concept indicating that the label of L23 is L33.

5. Accommodation

In the previous section we saw how ANA can use its mapping and error recovery methods to do unfamiliar tasks. Though ANA's performance on the tasks it has tried is adequate, the first time it 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. In this section we will see how ANA makes the transformation of unfamiliar to familiar tasks permanent. 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. This section has the same structure as Section 4. I will first talk about how ANA preserves contact and mapping knowledge. Then I will talk about how ANA patches methods that prove inadequate for some new task.

5.1 Method Building

In Section 4.1, I described what ANA does when faced with an unfamiliar task: it makes contact (via method description productions) with methods that might be used analogically, maps the pseudo-goal-description of each of these methods into the description of its actual goal, evaluates the candidate methods and selects one to try, and asserts, for each constraint, two pseudo-value containing constraints, one marked "constraint possible", the other marked "constraint given". ANA is then ready to use the method it has selected. However, if ANA is to avoid having to repeat these steps on subsequent occasions when it is called upon to do the same task, 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 small problem. Since it has not yet tried to use the method it has selected, it has no idea of 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 -- a list that will match the current goal element when that element is marked "goal achieved". The action side builds the production described above. Thus if ANA is able to accomplish its task, a production that makes the task permanently familiar will be built; if ANA fails, the production is not built. It should be noted that if ANA tries to use a method for a particular task, and the method turns out not to work, some useful knowledge may nevertheless have been gained. If in the course of using an ultimately unsuccessful method, subtasks are tried and accomplished, productions that map these subtasks into the submethods that succeeded will in fact be built. Thus, on subsequent occasions when ANA is called upon to do these subtasks, it will have the necessary knowledge. The production shown in Figure 5-1 is the production

```
p2-06   ( (act*1 object*2 (goal achieved =time)) & =w
        -->
         (<delete> =w) (act*1 object*2 (goal old =time))
         (<build>
            ( (=act =object (goal not-achieved =))
              (type =act (constraint given =time-a) wash)
              -(type =act (constraint possible >time-a) paint)
              (effect =act (constraint given =time-a) =effect surface)
              (state =effect (constraint given =time-a) clean)
              (location =object (constraint given =time-o) =location)
              (label =location (constraint given =time-o) L12)
              (type =object (constraint given =time-o) thing)
             -->
              (type =act (constraint given ((<quote> <bind>) =new-time)) paint)
              (type =act (constraint possible =new-time) paint)
              (map =act (concept true =object) (type paint) (type wash))
              (effect =act (constraint given =new-time) =effect surface)
              (effect =act (constraint possible =new-time) =effect surface)
              (location =object (constraint given =new-time) =location)
              (location =object (constraint possible =new-time) =location)
              (label =location (constraint given =new-time) L32)
              (label =location (constraint possible =new-time) L32)
              (map =location (concept true =object) (label L32) (label L12))
              (type =object (constraint given =new-time) table)
              (type =object (constraint possible =new-time) table)
              (map =object (concept true =object) (type table) (type thing))
              (color =effect (constraint given =new-time) red)
              (color =effect (constraint possible =new-time) red)
              (map =effect (concept true =object) (color red) (state clean)))))
```

Figure 5-1: A production that recognizes the task of washing the thing in L12

that ANA builds after it decides to use its "paint table" method to wash the thing in location L12 (Task2). The single condition element in this production will be satisfied only once -- when the goal element in working memory whose action link is "act*1" and whose object link is "object*2" is marked "goal achieved". The action side actually contains two actions (one more than I said). The first action marks the goal element "goal old"; the purpose of this action is to insure that if several productions are built, all of which match the same goal element, only the most recently built production will fire.[1] The other action simply builds a production that will fire whenever ANA is given the task of washing the thing in L12. The

[1] I give an example below of a case in which more than one production matching the same "goal achieved" element gets built. This happens whenever ANA tries a method and runs into a problem -- either an operator failure or an unexpected result -- due to a mutable attribute of the object it is acting on.

condition side of the production to be built will match the constraints given in the actual description of the task. The action side asserts all of the elements that must be asserted in order to make contact with the "paint table" method, plus those elements that must be in working memory in order for ANA to know that it is using its "paint table" method analogously.

After ANA has used a method (and submethods) analogously to accomplish a task, it has a number of new productions in its production memory that will enable it to accomplish the task much more easily (quickly) in the future. If ANA is given the same task, the production that causes the analogous method to be invoked will fire, and ANA will simply use that method. If that method generates a subgoal, another of the new productions -- one that enables a submethod to be used analogously -- will fire, and ANA will use the submethod. In many cases, this scheme works fine. There are, however, at least two situations in which it is inadequate. First of all, as we have seen, ANA in evaluating candidate methods sometimes discovers that in order to be able to use a method, it must first achieve some subgoal that satisfies a pre-condition of the method. If ANA is to avoid having to rediscover, each time it is given a task, that this subgoal must be achieved, it must include in the production it builds, the knowledge that achieving this subgoal is necessary. The second situation in which the basic scheme is inadequate is the situation in which ANA infers (or is told) that achieving some subgoal will not have the appropriate effect unless a different object description is substituted for the one asserted. In this case, ANA must build a production that recognizes the particular context in which the substitution is called for. When ANA decides to use a method that it knows has an unsatisfied pre-condition, it generates a subgoal to satisfy the pre-condition before it builds the production that associates the method with the current goal. If ANA achieves the subgoal, it generates a list containing each "constraint given" in working memory that is linked to the subgoal just achieved for which there is no corresponding element marked "constraint possible". Then it generates the two lists that it would ordinarily generate -- the one containing the condition side of the production to be built and the other containing the action side. Then it inserts the elements in the subgoal list at the front of the list containing the action side. Thus the production built, whenever it fires, will first assert the subgoal and will then assert the pseudo-goal-description that will make contact with the method. When ANA is given the task of painting the blue chair red (Task1) and finds that its "carry table" method can be used only if it can make the blue chair the top chair in the stack, it generates a cleartop goal. After the green chair is moved and the blue chair has become the top object in the stack, ANA generates a list containing all of the elements marked "constraint true" contained in the cleartop goal description. The production that ANA then builds to make contact with its "carry table" method is shown in Figure 5-2. After the task of carrying the blue chair is achieved, this production fires. The

```
p1-25   ( (act*53 object*2 (goal achieved =time)) & =w
        -->
         (<delete> =w) (act*53 object*2 (goal old =time))
         (<build>
            ( (=act =object (goal not-achieved =))
              (type =act (constraint given =time-a) carry)
              -(type =act (constraint possible >time-a) carry)
              (effect =act (constraint given =time-a) =effect location)
              (location =effect (constraint given =time-a) =location-effect)
              (label =location-effect (constraint given =time-a) L23)
              (location =object (constraint given =time-o) =location)
              (label =location (constraint given =time-o) L21)
              (color =object (constraint given =time-o) blue)
              (type =object (constraint given =time-o) chair)
             -->
              ((<quote> (<bind> =new-act)) =object (goal not-achieved (<quote> (<bind>))))
              (type =new-act (constraint given (<quote> (<bind> =new-time-a))) cleartop)
              (effect =new-act (constraint given =new-time-a)
                               (<quote> (<bind> =new-effect)) position)
              (position =new-effect (constraint given =new-time-a) (1))
              (type =act (constraint given ((<quote> <bind>) =new-time)) carry)
              (type =act (constraint possible =new-time) carry)
              (map =act (concept true =object) (type carry) (type carry))
              (effect =act (constraint given =new-time) =effect location)
              (effect =act (constraint possible =new-time) =effect location)
              (location =effect (constraint given =new-time) =location-effect)
              (location =effect (constraint possible =new-time) =location-effect)
              (label =location-effect (constraint given =new-time) L23)
              (label =location-effect (constraint possible =new-time) L23)
              (map =location-effect (concept true =object) (label L23) (label L23))
              (location =object (constraint given =new-time) =location)
              (location =object (constraint possible =new-time) =location)
              (label =location (constraint given =new-time) L32)
              (label =location (constraint possible =new-time) L32)
              (map =location (concept true =object) (label L32) (label L21))
              (type =object (constraint given =new-time) table)
              (type =object (constraint possible =new-time) table)
              (map =object (concept true =object) (type table) (type chair))))))
```

Figure 5-2: A production that recognizes the task of carrying a chair from L21 to L23 and that knows about the cleartop pre-condition

result is a production that fires whenever it is given the task of carrying a chair from location L21 to location L23.[1] When ANA generates a subgoal that contains the description of a new object, it tries to figure out if it should substitute some other object description for the one asserted; if it decides that it should, and the instructor concurs, it asserts a different object description. But whatever it decides, it remembers the decision. It builds a production that will fire whenever that subgoal is generated in the context of the goal it is trying to achieve. If, for example, ANA is given the task of washing the thing is L12 (Task2), the subgoal of carrying the paint can labeled red from L14 to L24 is generated. After ANA figures out that it should instead carry the water bottle from L25 to L24, it builds the production shown in Figure 5-3. It goes about building this production in much the same way that it goes about building the productions described above; that is, it collects the relevant constraints in working memory. The first condition element in the production in Figure 5-3 matches the goal element that is asserted by the production that watches for the assertion of subgoals containing new object descriptions; this element links the subgoal just generated to the current goal. The other condition elements are the set of stipulated constraints on the subgoal and the set of stipulated constraints on the goal. The action side of the production asserts the set of subgoal constraints, marked "constraint given", that are to be substituted for the original subgoal constraints, replaces each of the original subgoal constraints with a pair of constraints, one of which is marked "constraint possible" and the other "constraint given", and asserts a map element for each constraint.

5.2 Method Patching

In Section 4.2, I discussed ANA's way of dealing with problems that arise when the method it is using is either under-specified or over-specified. In some cases, the fixes described in that section not only solve the immediate problem, but also solve the problem whenever it subsequently arises. In the other cases, ANA must do some additional work to insure that it will be able to handle the problem effectively if it arises again. In discussing ANA's response to problems arising from under-specified methods, I distinguished between problems that ANA recognizes because of the failure of some operator and problems that it recognizes because of an unexpected result. Although this distinction is significant when discussing how ANA knows when it is in trouble, it is not significant when discussing how ANA fixes the problems. Therefore I will ignore the distinction in this section.

ANA responds in one of two ways to a problem that arises because it is using an

[1] The production will fire whether or not the chair to be carried is at the top of the stack in L21. If it is already at the top of the stack, the cleartop submethod will immediately discover that its goal is achieved and the carry method will take over.

```
p2-16  ( (replace sub-method (goal not-achieved =time)
                    (=act =object) (=new-act =new-object) =pox) & =w
          (type =new-act (constraint given =) =new-act-type)
          (effect =new-act (constraint given =) =new-effect =effect-type)
          (label =new-object (constraint given =new-time-o) red)
          (label =location (constraint given =new-time-o) L14)
          (type =new-object (constraint given =new-time-o) pntcan)
          (label =location-effect (constraint given =new-time-a) L24)
          (location =new-effect (constraint given =new-time-a) =location-effect)
          (location =new-object (constraint given =new-time-o) =location)
          (effect =act (constraint given =) =effect =)
          (state =effect (constraint given =time-a) clean)
          (type =object (constraint given =time-o) thing)
          -(= =object (constraint possible =time-o) =)
          -(= =act (constraint possible =time-a) =)
         -->
          (replace sub-method (goal achieved =time)
                    (=act =object) (=new-act =new-object) =pox) (<delete> =w)
          (type =new-act (constraint given =new-time-a) =new-act-type)
          (effect =new-act (constraint given =new-time-a) =new-effect =effect-type)
          (label =new-object (constraint given =new-time-o) water)
          (<delete> (label =new-object (constraint given =new-time-o) red))
          (label =new-object (constraint given 0) red)
          (label =new-object (constraint possible 0) red)
          (map =new-object (concept true =new-object) (label red) (label water))
          (label =location (constraint given =new-time-o) L25)
          (<delete> (label =location (constraint given =new-time-o) L14))
          (label =location (constraint given 0) L14)
          (label =location (constraint possible 0) L14)
          (map =location (concept true =new-object) (label L14) (label L25))
          (type =new-object (constraint given =new-time-o) watbot)
          (<delete> (type =new-object (constraint given =new-time-o) pntcan))
          (type =new-object (constraint given 0) pntcan)
          (type =new-object (constraint possible 0) pntcan)
          (map =new-object (concept true =new-object) (type pntcan) (type watbot))
          (label =location-effect (constraint given 0) L24)
          (label =location-effect (constraint possible 0) L24)
          (map =location-effect (concept true =new-object) (label L24) (label L24))
          (location =new-effect (constraint given 0) =location-effect)
          (location =new-effect (constraint possible 0) =location-effect)
          (map =new-effect (concept true =new-object)
                    (location =location-effect) (location =location-effect))
          (location =new-object (constraint given 0) =location)
          (location =new-object (constraint possible 0) =location)
          (map =new-object (concept true =new-object)
                    (location =location) (location =location)))
```

Figure 5-3: A production that remembers that "water bottle" is to "wash" as "paint can" is to "paint"

under-specified method. If the problem is due to some feature of the environment that can be changed (ie, if the problem is due to a mutable attribute of some object), then ANA simply builds a production that associates the action type of the method that it is using with a list containing the attribute and the value that the attribute must have in order for that action type to be appropriate; this production fires whenever ANA considers using a method with that action type. This fix is sufficient to enable ANA to deal effectively with the problem if it ever arises again. 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 that has to have that value does not, ANA knows that in order to use the method it is considering, it must generate a subgoal to satisfy this pre-condition.

ANA's other response to a problem that arises because it is using an under-specified method is more complex. If the problem is due to some feature of the environment that cannot be changed (ie, if the problem is due to an immutable attribute of some object), then 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. I described briefly, in Section 4.2, how ANA does this. 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. There is, of course, a much simpler way for ANA to overcome immutable attributes. ANA could build a production whose condition side matched the most recently asserted element in working memory and whose action side executed the operator it wanted to try. But this fix would not help ANA overcome the problem if it arose again at a later time. In order to provide a more permanent solution, ANA must inhibit the firing of the production that operates on the environment until it has had a chance to get the additional information that is necessary in order to determine if that operator is appropriate. Since ANA's only means of inhibiting the firing of an enabled production is to build a production that will be selected in preference to it during conflict resolution, and since the only elements that ANA can be sure will be in working memory when a production is enabled are just those elements that enable it, ANA builds a production with the same condition side as the production that operates on the environment and relies on the fact that, all other things being equal, more recently built productions are preferred in conflict resolution to older ones. The reason for making the condition side of the other production that is built (the production that executes a different operator) 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.

When ANA is asked to carry the box at the bottom of L11 to L23 (Task3), it first clears off the box and then attempts to @carry it. @carry fails, so ANA tries @push. @push fails, so ANA tries @cart. Since the box at the bottom of L11 is "really-heavy", @cart works. Three productions are shown in Figures 5-4 and 5-5. The production, "carry3", is the production in the "carry table" method that actually executes @carry. "p3-36" and "p3-37" are productions that ANA built when the @push operator failed. "p3-37" fires if "p3-36" finds that the weight of the object to be carried is something other than "heavy". In order to build these productions, ANA simply generates a list containing each element in working memory, marked "constraint given", that is linked to the current goal, and for which there is a corresponding element marked "constraint possible". Notice that in this case, ANA wants the constraints whose values are pseudo-values, since these are the constraints matched by the condition side of the production being masked.[1]

```
carry3  ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) carry)
          (effect =act (constraint given =time-a) =effect location)
          (location =effect (constraint given =time-a) =location-effect)
          (label =location-effect (constraint given =time-a) L23)
          (type =object (constraint given =time-o) table)
          (location =object (constraint given =time-o) =location)
          (label =location (constraint given =time-o) L32)
          (token =object (concept true =) =token1)
          (token =location (concept true =) =token2)
          (location =token1 (concept true >time) =token2)
          (label =token2 (concept true >time) L32)
          (label =other-object (constraint given =time-oo) L23)
          (token =other-object (concept true =) =token3)
          (composition =token3 (concept true >time) empty)
         -->
          (@carry =token1 =token3) )
```

Figure 5-4: One of ANA's "carry table" productions

[1] When ANA is using a method "directly" (ie, not analogously) and finds that it is under-specified, it generates the goal of doing whatever task it is doing by analogy. Since it may select the method it was using directly as the method to use analogously, ANA does have a way of patching its original methods.

```
p3-36   ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) carry)
          (effect =act (constraint given =time-a) =effect location)
          (location =effect (constraint given =time-a) =location-effect)
          (label =location-effect (constraint given =time-a) L23)
          (location =object (constraint given =) =location)
          (label =location (constraint given =) L32)
          (type =object (constraint given =) desk)
          (position =object (constraint given =) (3))
          (token =object (concept true =) =token1)
          (position =token1 (concept true =) (1))
          (location =token1 (concept true =) =token2)
          (token =location (concept true =) =token2)
          (label =token2 (concept true =) L21)
          (type =other-object (constraint given =time-oo) location)
          (label =other-object (constraint given =time-oo) L23)
          (token =other-object (concept true =) =token3)
          (composition =token3 (concept true =) empty)
         -->
          ((<bind> =new-act) =object (goal not-achieved (<bind>)))
          (type =new-act (constraint given (<bind> =new-time)) find)
          (effect =new-act (constraint given =new-time) (<bind> =new-effect) mind)
          (weight =new-effect (constraint given =new-time) really-heavy))


p3-37   ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =time-a) carry)
          (effect =act (constraint given =time-a) =effect location)
          (location =effect (constraint given =time-a) =location-effect)
          (label =location-effect (constraint given =time-a) L23)
          (location =object (constraint given =) =location)
          (label =location (constraint given =) L32)
          (type =object (constraint given =) desk)
          (position =object (constraint given =) (3))
          (token =object (concept true =) =token1)
          (position =token1 (concept true =) (1))
          (location =token1 (concept true =) =token2)
          (token =location (concept true =) =token2)
          (label =token2 (concept true =) L21)
          (type =other-object (constraint given =time-oo) location)
          (label =other-object (constraint given =time-oo) L23)
          (token =other-object (concept true =) =token3)
          (composition =token3 (concept true =) empty)
          (weight =token1 (concept true =) (@notany heavy))
         -->
          (@cart =token1 =token3))
```

Figure 5-5: A patch for an under-specified method

ANA does one more thing to help insure that it will not fall prey to the same problem in the future. Since ANA does not know in advance that it has an operator that will work, it builds a third production that associates the action type of the method it is using with a list containing the attribute and the value that the attribute must have in order for the original operator to work. Whenever a method with that action type is considered, this production will fire, and ANA will put the method in the category of "unlikely to work". Of course if the alternative operator that ANA tries does in fact work, the method belongs in the "will probably work" category. Therefore ANA builds a fourth production that will fire if the current goal is achieved; this production builds a production that associates the action type of the method being used with a list containing the attribute and the value that the attribute must have in order for the alternate operator to work. If the method works when the alternate operator is used, the production built by the fourth production neutralizes the effect of the third production.

As we have seen, if ANA is using its "find" method and the value of one of the expected effect constraints differs from the value of the corresponding percept, it asks the instructor whether it can ignore that expected effect. If the instructor indicates that the expected effect can be ignored, ANA asserts a concept containing the token-name given in the percept and the value given in the expected effect constraint. In order to produce a permanent fix, ANA simply builds a production that will assert that concept whenever it is given an object to find in the context of the current action type. For example, when ANA uses its "move box" method (see Figure 4-4) it expects that the location it will move things to is L33. The production in Figure 5-6 is the production that ANA built when it was moving the green chair and was told that it was fine to move the chair to some location other than L33.

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

```
p1-21   ( (=act =object (goal not-achieved =time))
          (type =act (constraint given =) find)
          (type =object (constraint given =) location)
          (composition =object (constraint given =) empty)
          (token =object (concept true =) =token1)
          (label =token1 (concept true >time) =value)
          -(label =token1 (concept true =) L33)
          (=main-act = (goal not-achieved <time))
          (type =main-act (constraint given =time-main) move)
          -(type =main-act (constraint possible =time-main) move)
         -->
          (label =token1 (concept true (<bind>)) L33)
          (map =object (concept true =object) (label L33) (label =value))
          (<build>
             ( ((=main-act = (goal (@any achieved old) =))
               (label =token1 (concept true =time-c) L33) & =scope
              -->
               ((<quote> <delete>) =scope) (label =token1 (concept old =time-c) L33))))
```

Figure 5-6: A patch for an over-specified method

not modify any of the productions comprising the method.

The question is: 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? Before trying to answer these questions, I want to briefly consider the distribution and effectiveness of ANA's knowledge. Then I will argue that ANA's only critical weakness is its lack of knowledge about how to select appropriate methods.

ANA's knowledge can be conveniently divided into three categories: (1) the initial, task-specific knowledge that ANA uses to manage its paint shop; (2) the knowledge ANA has that enables it to use its task-specific methods analogically; and (3) the new, task-specific knowledge that ANA acquires that enables it to extend its paint shop management capabilities. Figure 6-1 shows the number of productions in each of these categories. ANA starts with 57 productions that enable it to perform a few paint shop tasks. It has 238 productions that enable it to use its task methods analogically. The productions in the "knowledge nets" sub-category contain the knowledge of relationships among actions and among objects. The "setting up analogies" sub-category is the knowledge ANA needs to select and prepare to use a method on an unfamiliar task. The "using analogies" subcategory is the knowledge that enables ANA to maintain the integrity of its mappings and to determine their plausibility. The productions in the "under-specified recovery" and "over-specified recovery" sub-categories

contain the knowledge that ANA needs to recover from problems that arise as it uses its methods analogically. In the course of doing the four tasks described above, ANA built 141 productions; some of these associate a new task with an existing method, some extend its knowledge nets, and some patch methods that proved inadequate.

Initial methods	
Task methods	36
The "find" method	21
Analogy tools	
Knowledge nets	72
Setting up analogies	39
Using analogies	78
Under-specified recovery	40
Over-specified recovery	9
New productions	
Method association	49
Net extension	8
Method patching	84
Total	436

Figure 6-1: The distribution of ANA's knowledge (number of productions)

Figure 6-2 shows the time in cycles (ie, production firings) that ANA spent on each of the four tasks described above. For each task there are two columns of numbers: The columns labeled "Run1" show how long it takes ANA to perform unfamiliar tasks; the columns labeled "Run2" show how long it takes ANA to perform the same tasks once they have become familiar. A rough measure of the effectiveness of ANA's learning strategy can be obtained by comparing the amount of time ANA spends using its initial methods with the total time that it spends on a task. If ANA had been provided with a set of methods for doing the four tasks, then (as Figure 6-2 shows) it would have taken ANA about 151 cycles to do Task0, 225 cycles to do Task1, 120 cycles to do Task2, and 302 cycles to do Task3. Thus it takes ANA about 3 to 6 times as long to do an unfamiliar task as it would take it to do a familiar task of comparable difficulty. It takes ANA about 1.3 to 1.8 times as long to do a task that it can do in virtue of its accommodation capability as it would take it to do the same task with a set of methods designed for the task.

Given ANA's performance on the four 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

	Task0		Task1		Task2		Task3	
	Run1	Run2	Run1	Run2	Run1	Run2	Run1	Run2
Initial methods								
Task methods	30	25	56	35	23	19	68	46
The "find" method	140	126	273	190	116	101	314	256
Analogy tools								
Knowledge nets	75	0	254	0	125	0	269	0
Setting up analogies	65	0	236	0	92	0	228	0
Using analogies	171	47	497	143	243	65	546	155
Under-specified recovery	0	0	31	0	28	0	123	0
Over-specified recovery	0	0	28	7	8	0	52	29
New productions								
Method association	3	3	8	7	4	4	9	7
Net extension	0	0	6	0	0	0	6	0
Method patching	0	3	9	16	4	7	23	19
Total	484	204	1398	398	643	196	1638	512

Figure 6-2: ANA's performance on four tasks (number of cycles)

at least one of these methods is almost adequate for each unfamiliar task with which the system will be faced, 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) The method being used can be under-specified; so the system will have to learn what the additional pre-conditions on the method are. (2) The method being used can be over-specified; so the system will have to learn what constraints to ignore. (3) 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. The mechanisms required to solve all three problems are very 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 would 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

Anderson, J. R., P. J. Kline and C. M. Beasley Jr. Complex learning processes. Technical Report. Department of Psychology, Yale University, 1978a.

Anderson, J. R., P. J. Kline and C. M. Beasley Jr. A Theory of the acquisition of cognitive skills. Technical Report. Department of Psychology, Yale University, 1978b.

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

Forgy, C. and J. McDermott. OPS, a domain-independent production system language. **IJCAI, 5**, 1977.

McDermott, J. Some strengths of production system architectures. **Proceedings of NATO ASI on Structural/Process Theories of Complex Human Behavior.** Sijthoff, 1978.

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.

Moore, J. and A. Newell. How Can Merlin Understand? In L. Gregg (ed), **Knowledge and Cognition.** Lawrence Erlbaum Associates, 1973.

Newell, A. Knowledge representation aspects of production systems. **IJCAI, 5**, 1977.

Rychener, M. Control requirements for the design of production system architectures. **Proceedings of the Symposium on AI and Programming** (SIGART/SIGPLAN), 1977.

Sussman, G. J. **A Computer Model of Skill Acquisition.** American Elsevier, 1975.