23

Some New Directions in Robot Problem Solving

R. E. Fikes, P. E. Hart and N. J. Nilsson
Artificial Intelligence Center
Stanford Research Institute

Abstract
For the past several years research on robot problem-solving methods has centered on what may one day be called 'simple' plans: linear sequences of actions to be performed by single robots to achieve single goals in static environments. Recent speculation and preliminary work at several research centers has suggested a variety of ways in which these traditional constraints could be relaxed. In this paper we describe some of these possible extensions, illustrating the discussion where possible with examples taken from the current Stanford Research Institute robot system.

1. INTRODUCTION

Background

A major theme in current artificial intelligence research is the design and construction of programs that perform robot problem solving. The usual formulation begins with the assumption of a physical device like a mechanical arm or a vehicle that can use any of a preprogrammed set of actions to manipulate objects in its environment. The generic task of the robot problem solver is to compose a sequence of these actions (or *operators*) that transforms the initial state of the environment (or problem domain) into a final state in which some specified goal condition is true. Such a sequence is called a *plan* for achieving the specified goal.

Most previous work in robot problem solving – for example, the work of Green (1969), Fikes and Nilsson (1971), and Winograd (1971) – has been limited at the outset by the assumption of a certain set of simplifying ground rules. Typically, the problem environment is a dull sort of place in which a single robot is the only agent of change – even time stands still until the robot moves. The robot itself is easily confused; it cannot be given a second problem until it finishes solving the first, even though the two problems may be related in some intimate way. Finally, most robot systems cannot yet generate plans containing explicit conditional statements or loops.

In this paper we wish to consider some possibilities for relaxing these ground rules. Our suggestions are of course tentative, and perhaps not even entirely original. We hope, nevertheless, that they will illuminate some of the issues by serving as thought-provoking examples of what might be done next in this interesting research area. We specifically exclude from our discussion comments about present and future trends in robot perception (vision, range-finding, and the like), except to make the obvious remark that advances in perceptual abilities will ease the problem-solving burden and vice versa.

As an aside to the reader, we admit to having difficulty in discussing our ideas in an abstract setting; we find we understand the ideas ourselves only when we see examples. Accordingly, we elected to couch our suggestions in the language and framework of a particular robot problem solver, STRIPS (Fikes and Nilsson 1971, Fikes, Hart and Nilsson, in press), that has been under development at Stanford Research Institute. Our discussion, therefore, takes on the tinge of being a critique of this particular system, but we hope that some of the ideas have a more general interpretation. To provide the necessary background for the reader unfamiliar with the STRIPS system, we will try to give in the next few paragraphs as brief and painless a summary as possible.

A summary of STRIPS

A problem environment is defined to STRIPS (STanford Research Institute Problem Solver) by giving two different kinds of information: a model consisting of a set of statements describing the initial environment, and a set of operators for manipulating this environment. An operator is characterized by a precondition statement describing the conditions under which it may be applied, and lists of statements describing its effects. Specifically, the effect of an operator is to remove from the model all statements matching forms on the operator's 'delete list', and to add to the model all statements on the operator's 'add list'. All statements are given to STRIPS in the predicate-calculus language.

Once STRIPS has been given an initial model and a set of operators, it may be given a problem in the form of a goal statement. The task for STRIPS is to find a sequence of operators transforming the initial model, or state, into a final state in which the goal is provable. STRIPS begins operation by attempting to prove the goal from the initial model. If the proof cannot be completed, STRIPS extracts a 'difference' between the initial model and the goal indicating a set of statements that would help to complete the proof. It then looks for a 'relevant operator' that will add to the model some of the statements in the difference. (For example, a difference may include a desired robot location; this difference would be reduced by a GOTO operator.) Once a relevant operator has been selected, its preconditions constitute a subgoal to be achieved, and the same problem-solving process can be applied to it. When a precondition subgoal is provable in the state under consideration, then the subgoal has been achieved and the operator's effects description is used to produce a new state. This process of forming new subgoals and new states continues until a state is produced in which the original goal is provable; the sequence of operators producing that state is the desired solution.

After STRIPS produces a solution to the specific problem at hand, the solution is *generalized* and stored in a special format called a *triangle table.* The generalization process replaces specific constants in the solution by parameters so that the plan found by STRIPS will be applicable to a whole family of tasks including the special task of the moment. The generalized plan can then be used as a component macro-action in future plans. Such a macro-action is called a MACROP and is also used to monitor the execution of a plan. Roughly, our execution monitor, PLANEX, has the useful ability to find an appropriate new instance of a general MACROP if it finds that the instance being executed fails to work for some reason.

Details of our system for learning and executing plans will be published (Fikes, Hart and Nilsson, in press); we shall give a brief explanation of the triangle table format here, since part of the discussion to be presented in Section 5 depends on it. (The reader may want to defer reading the rest of this section until later.) An example of a triangle table is shown in figure 1 for the case of a plan with three component steps. The cells immediately below each operator contain the statements added by that operator; we have used the notation A_i to represent these add lists. We retain in the cells below the top cell in a column just those statements that are not deleted by later operators. For example, $A_{1/2}$ in cell (3, 2) represents the statements of A_1 that remain in the model after the application of operator OP_2; similarly, $A_{1/2,3}$ represents the statements in $A_{1/2}$ that survive in the model after the application of OP_3.

1	PC_1	OP_1			
2	PC_2	A_1	OP_2		
3	PC_3	$A_{1/2}$	A_2	OP_3	
4		$A_{1/2,3}$	$A_{2/3}$	A_3	GOAL
	1	2	3	4	

Figure 1. An example of a triangle table.

The left-most column of the triangle table contains certain statements from that model existing before any of the operators were applied. STRIPS, we recall, must always prove the preconditions of an operator from any given model before the operator can be applied to that model. In general, the model statements used to establish the proof arise from one of two sources:

either they were in the initial model and survived to the given model, or they were added by some previous operator. Statements surviving from the initial model are listed in the left-most column, while statements added by previous operators are among those listed in the other columns. By way of example, consider OP_3. In general, some of the statements used to support the proof of its preconditions were added by OP_2, and are therefore part of A_2; some of the statements were added by OP_1 and survived the application of OP_2, and are therefore included in $A_{1/2}$; finally, some of the statements existed in the initial model and survived the application of OP_1 and OP_2, and these statements comprise PC_3.

The triangle table format is useful because it shows explicitly the structure of the plan. Notice that all the statements needed to establish the preconditions of OP_i are contained in Row i. We will call such statements *marked* statements. By construction, all the statements in the left-most cell of a row are marked, while only some of the statements in the remainder of the row may be marked.

Obviously, an operator cannot be executed until all the marked statements in its row are true in the current state of the world. This alone is not a sufficient condition for execution of the rest of the operators in the plan, however. To point up the difficulty, suppose all the marked clauses in Row 2 are true, but suppose further that not all the marked statements in cell (3, 2) are true. We know that OP_2 can itself be executed, since its preconditions can be proven from currently true statements. But consider now the application of OP_3. Since the marked statements in cell (3, 2) are not true, it cannot be executed. Evidently, OP_1 also should have been executed.

The preceding example motivates an algorithm used by PLANEX. The algorithm rests on the notion of a *kernel* of a triangle table, the *i*th kernel being by definition the unique rectangular subtable that includes the bottom left-most cell and row *i*. In its simplest form, the PLANEX algorithm calls for executing OP_i whenever kernel *i* is the highest numbered kernel all of whose marked statements are true. The reader may want to verify for himself that this algorithm avoids the difficulty raised in the previous paragraph.

With this sketchy introduction to the STRIPS system, we can now proceed to more speculative matters of perhaps greater interest.

2. MULTIPLE GOALS

Multiple goals and urgencies

Useful applications of robots may require that the robot system work toward the achievement of several goals simultaneously. For example, a Martian robot may be performing some life detecting experiment as its main task and all the while be on the lookout for rock samples of a certain character. We should also include the possibility of 'negative goals': our Martian robot might be given a list of conditions that it must avoid, such as excessive battery drain and being too close to cliffs.

There are several ways in which to formulate task environments of this sort. In our work with STRIPS, a goal is given to the system in terms of a single predicate calculus wff (well-formed formula) to be made true. We might define multiple goals by a set of goal wffs, each possessing an 'urgency value' measuring its relative importance. Wffs having negative urgency values would describe states that should be avoided. Now we would also have to define precisely the overall objective of the system. In the case of a single goal wff, the objective is quite simple: achieve the goal (possibly while minimizing some combination of planning and execution cost). For an environment with multiple goals, defining the overall objective is not quite so straightforward, but quite probably would include maximizing some sort of benefit/cost ratio. In calculating the final 'benefit' of some particular plan one would have to decide how to score the urgencies of any goal wffs satisfied by the intermediate and final states traversed by the plan. In any case the essence of calculating and executing plans in this sort of task environment would entail some type of complex accounting scheme that could evaluate and compare the relative benefits and liabilities of various goals and the costs of achieving them.

Planning with constraints

There is a special case of the multiple goal problem that does not require complex accounting, yet exposes many key problems in a simplified setting. This special case involves two goals, one positive and one negative. The objective of the system is to achieve the single positive goal (perhaps while minimizing search and execution costs) while avoiding absolutely any state satisfying the negative goal. Thus, we are asked to solve a problem subject to certain constraints – some states are *illegal*.

Many famous puzzles such as the 'Tower of Hanoi Puzzle' and the 'Missionaries and Cannibals Problem' can be stated as constraint problems of this sort. Actually, the legal moves (operators) of these puzzles are usually restricted so that illegal states can never even be generated. But such restrictions on the preconditions of the operators are often difficult and awkward to state. Furthermore, if new constraints are added from time to time (or old ones dropped), the definitions of the operators must be correspondingly changed. We would rather have simple operator preconditions that *allow* operators to be applied even though they might produce illegal states. With this point of view we must add a mechanism that can recognize illegal states and that can analyse *why* they are illegal so that search can be guided around them.

Let us consider a simple example task using the environment of figure 2. Three rooms are connected by doorways as shown and the robot is initially in room R3. The robot can move around and push the boxes from one place to another. Let us suppose that the positive goal is to get one of the boxes into room R1. The negative goal is a box and the wedge, W1, in the same room; thus any state with this property is illegal. For this example we can consider two operators: PUSHRM(BX, RX, DX, RY) pushes object BX from RX into adjacent room RY through connecting doorway DX. A precondition of PUSHRM is that the robot and object BX be in room RX. GOTORM(RX, DX, RY) takes the robot from room RX into adjacent room RY through connecting doorway DX. We do not complicate the precondition of PUSHRM with such a requirement as 'If BX is a box, RY cannot already contain a wedge,' or the like.

We must now consider what modifications to make to STRIPS so that it does not generate a plan that traverses an illegal state. One obvious necessity is to test each new state produced to see if it is illegal, that is, to see if the negative goal wff can be proved in that state. If the state is illegal, the search for a plan must be discontinued along that branch of the search tree and taken up somewhere else.

In problem-solving systems such as STRIPS that use the GPS means-ends strategy, it will not do merely to discontinue search at points where illegal states are reached. We must also extract information about why the state is illegal so that other operators can be inserted earlier in the plan to eliminate undesirable features of the state.

In our example of figure 2, STRIPS might first decide to apply the operator PUSHRM(BOX1, R3, D3, R1). This application results in an illegal state. If we merely discontinue search at this point, STRIPS might next decide to apply PUSHRM(BOX2, R2, D1, R1) whose precondition requires first the application of GOTORM(R3, D2, R2). But PUSHRM again results in an illegal state. Ultimately there will be no place in the search tree left to search and STRIPS will fail to find a solution.

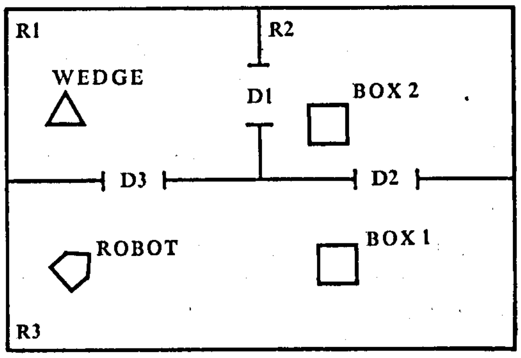


Figure 2. Robot environment for a constraint problem.

Thus, we need an analysis that is able to pursue a chain of reasoning such as the following:

(1) We have just applied PUSHRM(BOX1, R3, D3, R1), resulting in an illegal state.

(2) The reason the state is illegal is because it contains the two statements

INROOM(R1, W1)

INROOM(R1, BOX1).

They were used in the proof that the state was illegal.

(3) The operator PUSHRM is responsible for the occurrence in this state of the statement

INROOM(R1, BOX1).

Presumably the means-ends analysis technique had a reason for applying this operator and probably it was to add this statement.

(4) The other statement INROOM(R1, W1) was in the previous state since it was not added by PUSHRM.

(5) Therefore, let us add – just to this particular application of the operator PUSHRM – the additional precondition ~INROOM(R1, W1).

Thus, we would reconsider the subgoal of proving PUSHRM's preconditions in the node just above the one containing the illegal state. If there are several candidate statements whose negations could be added to the preconditions, we could create alternative nodes for each. Also, we could reconsider the whole chain of ancestor nodes containing PUSHRM's preconditions in their goal lists so that remedial action could be taken earlier in the plan.

In our example, we merely create a subgoal of getting W1 out of room R1 and, say, into room R2. This subgoal is easily achieved by pushing W1 into room R2 creating another illegal state. But this PUSHRM will also have its preconditions automatically elaborated to require that BOX2 be out of room R2. Ultimately, the plan GOTORM(R3, D2, R2), PUSHRM(BOX2, R2, D2, R3), GOTORM(R3, D3, R1), PUSHRM(W1, R1, D1, R2), GOTORM(R2, D2, R3), PUSHRM(BOX1, R3, D3, R1) would be constructed.

It should not be too difficult to modify STRIPS so that it could deal with negative goals in this manner. Further modifications might also be desirable. For example, at the time a relevant operator is selected, we might want to perform checks on it to see if it can be applied at all in light of the constraints. First we would ask whether its precondition formula alone implied the negative goal wff. There would be no point in working towards a state satisfying a subgoal precondition wff if that very wff implied a negative goal. Then we would ask whether the operator's add list alone implied a negative goal. These tests would indicate whether a relevant operator ought to be prohibited before STRIPS wastes time in planning to apply it.

Certain additional complications surround the use of previously learned plans or MACROPS by STRIPS during the planning process. STRIPS would have to check the legality of each intermediate world state produced by the components of a MACROP. If an illegal state is traversed, STRIPS would have the choice of either working around this illegal state to achieve the desired results of the entire MACROP or of planning to eliminate undesirable features before applying any of the MACROP. Decisions about the proper strategy will depend on our experience with systems of this sort.

Execution strategies

So far we have dealt only with questions concerning *planning* under constraints. After a plan is generated, our robot system must execute it, and during the execution we must avoid falling into illegal states.

Our first concern is with the process of plan generalization. STRIPS may have produced a perfectly fine specific plan that traverses no illegal states, but it could happen that its generalization has instances that do traverse illegal states. This difficulty may not be troublesome as concerns use of the MACROP in later planning, since STRIPS presumably will insure that none of these bad instances are used. During execution of the MACROP, however, we will have to make sure that our instantiation process does not select one of the bad instances.

Even if the same instance of the plan is executed as was planned, there is the added difficulty caused by new information discovered about the world during execution. While this new information might not affect at all the possibility of carrying out the original plan, it might inform us that certain of the states to be traversed are illegal. Suppose, for example, that STRIPS did not know of the existence of wedge W1 in the task of figure 2 and planned merely to push BOX1 into room R1. During execution of this plan (but, say, before BOX1 enters room R1), let us suppose that the robot discovers the wedge. What is to be done? Obviously STRIPS must generate a new plan; to do this, the executive system must have a mechanism for recognizing illegal states and for recalling STRIPS.

In certain task environments there might be negative goals that represent constant constraints for all of the tasks ever to be performed. Instead of having to plan to avoid these illegal states every time a new task is confronted, it may be more efficient to build up a list of operator instances whose application is prohibited whenever certain conditions obtain. (For example, don't walk on the train tracks when a train is coming.) STRIPS would then check each operator application to check its legality, and the executive would do likewise. In the executive system, we could regard this activity as a sort of *inhibition* of illegal operators. If a planned operator is inhibited, presumably STRIPS would be called to generate a new plan.

3. DYNAMIC ENVIRONMENTS

Introductory remarks

The STRIPS problem-solving system, as well as the associated execution routines, were designed to work in a stable, relatively static environment. Nothing changed in the environment unless the robot system itself initiated the change. There were no independent, ongoing events; for such environments we did not need the concept of time. Furthermore, the actions that the robot could perform (pushing boxes and moving about) had effects that could quite easily be represented by the simple add and delete lists of the STRIPS operators.

To progress from the simple environment in which STRIPS plans are conceived and executed to a more realistic, dynamic environment requires some new concepts and mechanisms. Some of these have already received attention at SRI and elsewhere. We hope here to discuss some of these requirements and to point out what we think might be profitable approaches.

First, we shall deal with those problems caused by the fact that the ultimate effect of a robot action might not be easily represented by a STRIPS add-list formula. It may require, instead, a computation performed by a process that simulates some aspect of the environment. Next we shall consider the special problems caused by independent processes going on in the environment (perhaps, for example, another independent robot). Last, we shall give a brief discussion of how we might introduce the concept of time. All of our comments are highly speculative and hopefully will raise questions even though they will answer few.

Representation of complex actions

In a world that is relatively 'uncoupled,' the effects of a robot action can be described simply by the STRIPS add and delete lists. The effect of the robot going to place A is to add the wff AT(ROBOT, A) and to delete any wffs saying that the robot is anywhere else. In such a simple world it is unnecessary to inquire in each case whether an action has special side effects or perhaps touches off a chain of events that affect other relations. Such side effects that do occur can simply be taken care of in the add and delete lists. The STRIPS representation of the effects of an action is an example of what we shall call an *assertional* representation. But in more highly coupled worlds, the ultimate effects of actions might depend in a complex way on the initial conditions. These effects might in fact best be modeled by a computation to be performed on the world model representing the initial conditions. We shall say that these effects are modeled by a *procedural* representation.

Several AI researchers have already stressed the need for procedural representations. Winograd, for example, in his BLOCKS program (1971), made use of features in the PLANNER (Hewitt 1971) language to represent actions as procedures. The *consequent theorems* of PLANNER are much like STRIPS operators in that they name a list of relations that are established by an action and also specify the preconditions for the action. Those effects of the action that are explicitly named are represented assertionally as in STRIPS, but side effects and preconditions are represented procedurally. There is a definite order in which preconditions are checked, and there are provisions for directions about what should be done when failures are met. *Antecedent theorems* and *erase theorems* allow more indirect effects of the action to be computed in a procedural fashion.

We note that, even in PLANNER, the named effects of an action are represented assertionally. Assertional representations have a distinct advantage for use in planning since they permit straightforward mechanisms (such as PLANNER's pattern-directed invocation) for selecting those actions most relevant for achieving a goal. It would seem difficult to employ the strategies of means-ends analysis or chaining backwards if the effects of actions were completely hidden in procedural representations.

The assertional form of the STRIPS operators permitted us to store plans in a convenient form, *triangle tables*. These tables reveal the structure of a plan in a fashion that allows parts of the plan to be extracted later in solving related problems. It is not obvious whether one could find a similarly convenient format if operators were represented procedurally rather than assertionally. One possibility is to use operators represented assertionally as rough models for more precisely defined operators represented procedurally. High level planning could be accomplished as it now is by STRIPS, and then these plans could be checked using the more accurate procedural representations.

Independent processes

In this section we shall consider some complications that arise if there are other independent agents in the world that can perform actions on their own. For example, there might be other robots (with goals different, if not inimical, to our robot). In such a case we would need to compute plans that consider explicitly the possible 'moves' of the other agents. The other agents might not necessarily be governed by goal-oriented behaviour but might merely be other entities in nature with power to cause change, for example lightning, falling rocks, the changing seasons, and so on. In general, our robot system will not be able to predict perfectly the effects of these agents in any given situation. Thus, it might need to use game trees and minimaxing techniques. Here, though, we want to consider just the special case in which our robot system does have perfect information about what other agents will do in any given situation. Even this special case poses some difficult questions, and we think progress can be made by answering some of them first.

Before dealing with these questions, we want first to remark that what is independent and what is causal (that is, caused by our robot) we regard mainly as a matter of definition. It might, for example, be convenient to regard all but the *immediately* proximate effects of a robot action as effects caused by an independent (although perhaps predictable) agent. For example, we could take the view that the only immediately proximate effect of removing the bottom block from a tower of blocks is that the bottom block is no longer in the tower. An independent agent then is given the opportunity (which it never avoids) to act, making the other blocks fall down. Although such a division between causal and independent effects in this case sounds extreme, it may in fact have some computational advantages. At the other extreme of the spectrum of actions come those that indisputably are most conveniently thought of as being performed by an independent agent, say a human or another robot. If the second robot were constructed by the first one, the actions of the offspring could conceivably be regarded as complex effects caused by the parent, but this interpretation seems obviously unwieldy.

Without other agents of change in the world, nothing changes unless the robot initiates the change. The robot has a set of actions it can execute and, for any initial situation, it can predict the effects of each of them. The 'physics' of such a world are simply described by the (assertional or procedural) representation of the robot's actions. With other agents present the set of physical laws is more complex. Other agents cause changes, and we assume that we can predict them. We need a representational system (again either procedural or assertional) to describe the conditions under which these changes takes place and what the changes are.

An especially interesting situation arises when our robot system can take action to avert a change that would ordinarily take place if the robot were passive. An example is given by the rule: 'The robot will be crushed unless it gets out of the path of a falling rock.' Thus, in our dynamic world populated by other agents (some of which may merely be fictitious agents of nature) we have two main types of rules for describing the 'physics' of the world. Both descriptions are relative to certain specified preconditions that we presume are met:

(1) A relation R will *change* (that is, its truth value will change) if we perform an action a; otherwise it will remain the same (all other things being equal).

(2) A relation R will *stay the same* if we perform some action a; otherwise it will change (all other things being equal).

These two types of rules can be used as justification either to perform an action or to inhibit one. (Of course, as we learned in Section 2, it is only in the case of having multiple goals that it makes sense to speak of 'inhibiting' an action. An inhibited action is one that we have decided not to execute since, simply put, it does more harm than good.) Whether we perform or inhibit an action for each rule depends on whether our goal is to change or maintain the relation R. If we further divide the class of goals into two types, good or positive goals and bad or negative ones, we get eight different kinds of resulting tactics. These are listed, with an example of each, in the chart of table 1.

The split from four to eight tactics depends on whether we choose to distinguish between positive and negative goals. Such a distinction may prove unimportant, and we make it here primarily because it allows a nice correspondence to certain behavioral paradigms explored by psychologists. (There is considerable neurophysiological evidence indicating two separate motivational systems in animals: a positive or pleasure system with foci in the lateral hypothalamus and a negative or pain system with foci in the medial hypothalamus.)

The actions used by STRIPS (as well as most other robot problem solvers) correspond to the tactic 'excite action to bring reward.' The main problems

Table 1. Eight tactics for a dynamic environment.

Physics	Goal Type		Tactic	Example
R will change if we perform *a*; otherwise it will stay the same.	We want *R* to change . . .	toward a positive goal.	Excite action to bring reward.	Insert dime in slot to get candy.
		away from a negative goal.	Excite action to escape punishment.	Leave a room having a noxious odour.
	We want *R* to be maintained . . .	at a positive goal.	Inhibit action to maintain reward.	Do not spit out a fine wine.
		against changing toward a negative goal.	Inhibit action to avoid punishment.	Do not step in front of a moving automobile.
R will stay the same if we perform *a*; otherwise it will change.	We want *R* to change . . .	toward a positive goal.	Inhibit action to bring reward.	Do not leave restaurant after having just ordered dinner.
		away from a negative goal.	Inhibit action to escape punishment.	Do not run when out of breath.
	We want *R* to be maintained . . .	at a positive goal.	Excite action to maintain reward.	Follow the Pied Piper.
		against changing toward a negative goal.	Excite action to avoid punishment.	Save for a rainy day.

posed by a dynamic world for a planning system are seen to be those stemming from the need to be able to construct plans using the other seven tactics as well. Two of the present authors, in collaboration with Ann Robinson, have given some preliminary thought to matters relating to the *execution* of action chains containing all eight types of tactics (Hart, Nilsson and Robinson 1971). We expect that the matter of designing a combined planning and execution system to operate in a dynamic world will present some very interesting problems indeed.

Time

When dealing with a dynamic environment, it becomes increasingly difficult to ignore explicit consideration of the concept of time. We would like a robot system to be able to respond appropriately to commands like:

Go to Room 20 *after* 3:00 p.m., and *then* return *before* 5:00 p.m.
Go to Room 20 three times a day.
Go to Room 20 and *wait until* we give you further instructions.
Leave Room 20 *at* 4:00 p.m.

We would also like the robot system to be able to answer questions dealing with time. For example: 'How many times did you visit Room 20 before coming here?' (See the recent article by Bruce (1972) for an example of a formalism for a question answering system dealing with time.)

A straightforward way to approach some of these issues is to add a time-interval predicate to an assertional model. Thus TIME(A, B) means that it is after A and before B. Whenever the system looks at an external clock, then presumably A and B become identical with the clock time. The model would then need some mechanism for continually revising A and B. A growing spread between A and B would represent the known inaccuracy of the model's internal clock.

For the moment we might for simplicity assume that time stands still while the robot system engages in planning. That is, we assume that the time required for planning is insignificant compared to the time duration of other events such as robot actions. (Incidentally, this assumption is manifestly not true as regards the present SRI robot system.) We can begin to deal with time by including in the operator descriptions the effect the operator has on the time predicate. (We must do the same for descriptions of the effects of independent agents.) Thus if the operator GOTHRUDOOR takes between three and five units of time, its effect on the predicate TIME(A, B) is to replace it by the predicate TIME(A+3, B+5).

We might also want to have a special operator called WAIT(n) that does nothing except let n units of time pass. With these concepts it will probably be a straightforward matter to plan tasks such as 'Be in Room 20 after 3:00 p.m. but not before 5:00 p.m.' The system would calculate the maximum length of time needed to execute the needed actions and would insert the appropriate WAIT operator somewhere in the chain if needed. Since much planning will probably be done without reference to time at all; it will be most convenient to ignore explicit consideration of time unless the task demands it.

We suspect that allowing time to progress during planning will present many problems. Then the planner must operate with world models that will not 'stand still'. For certain time-dependent tasks; such as 'Meet me in Room 2 at 3:00 p.m.', the planner will have to be able to coordinate successfully the amounts of time spent in planning and task execution.

4. MULTIPLE OUTCOME OPERATORS

One of the interesting properties of a robot system is the inherent incompleteness and inaccuracy of its models of the physical environment and, in most cases, of its own action capabilities. This property implies a degree of indeterminism in the effects of the system's action programs and leads one to consider including in the planning mechanism consideration of more than one possible outcome for an operator application. We might like to model certain types of failure outcomes, such as a box sliding off the robot's pushbar as it is being pushed. We might also like to model operators whose primary purpose is to obtain information about the physical environment, such as whether a door is open or whether there is a box in the room. Munson (1971) and Yates (private communication) have also discussed multiple outcome operators.

We can extend the STRIPS operator description language in a natural way to provide a multiple outcome modeling capability as follows: each description can have *n* possible outcomes each defined by a delete list, an add list, and (optionally) a probability of the outcome. For example, an operator that checked to see if a door was open or closed might have the following description:

```
CHECKDOOR(DX)
  PRECONDITIONS
    TYPE(DX, DOOR)
    NEXTTO(ROBOT, DX)
  outcome 1                      outcome 2
  DELETE LIST                    DELETE LIST
    NIL                            NIL
  ADD LIST                       ADD LIST
    DOORSTATUS(DX, OPEN)           DOORSTATUS(DX, CLOSED)
  PROBABILITY                    PROBABILITY
    0.5                            0.5
```

We may also want to provide advice about when information gathering operators ought to be applied. For example, it would be inappropriate to apply CHECKDOOR if the robot already knew the value of DOORSTATUS. This advice might be supplied by including in the preconditions a requirement that the information to be gathered is not already known before the operator is applied. This requirement would insure that the operator description indicates that information is being added to the model rather than being changed in the model and would therefore allow the planner to distinguish CHECKDOOR from an operator that opened closed doors and closed open doors.

Such an 'unknown' requirement would be a new type of precondition for STRIPS, since what is required is not a proof of a statement but a failure both to prove a statement and to prove the statement's negation. This requirement is analogous to the semantic meaning of the THNOT predicate in PLANNER (Hewitt 1971); that is, THNOT $p(x)$ is true if the PLANNER interpreter cannot prove $p(x)$ with some prespecified amount of effort. A capability of handling such preconditions could be added to STRIPS in a natural manner by defining new syntax for the preconditions portion of an operator description and by adding new facilities to the goal testing mechanism in STRIPS to attempt the required proofs.

Let us consider the planner's search space when multiple outcome operators are used. We assume a search tree where each node represents a state and each branch from a node represents the outcome of an operator applied to the node. This is a standard AND/OR tree in which the outcomes from different operators are OR branches and the outcomes from a single operator are AND branches. Any plan is a subtree of this tree consisting of the initial node of the tree and for each node of the plan exactly those offspring nodes produced by the outcomes of a single operator; hence, from each non-terminal node of a plan there will emanate one set of AND branches. Each terminal node must correspond to a world model satisfying the goal wff if the plan is to be successful.

When such a plan is executed, each branch in the tree corresponds to an explicit conditional test. The test determines what was the actual outcome of the action routine and the corresponding branch of the remainder of the plan is executed next. Thus the problem of building into plans explicit conditional statements does not appear difficult; providing explicit loops looks considerably harder.

If probabilities of the different outcomes of operators are defined, then the probability of occurrence of any node in a plan tree (during execution of the plan) might be considered to be the product of the probabilities of the branches connecting the node with the initial node of the tree.

One of the interesting issues facing the planning program is when to stop. One certainly would have the program stop when it achieved absolute success, in that a plan existed each of whose terminal nodes represented a state that satisfied the task; for such a plan each of its operators could produce any of the described outcomes and the plan would still lead to success. Similarly, the planner would stop when it reached an absolute failure; that is, when the tree had been fully expanded and no terminal node of any plan satisfied the task; in such a situation we know that none of the plans in the tree will lead to success no matter what the outcomes of the operators. A node has been fully expanded when all possible offspring nodes have been included in the tree. Thus, terminal nodes in a fully expanded tree have no possible offspring. To make the concept of a fully expanded node more practical, one may assume that only 'relevant operators' are considered for producing offspring.

But what about other situations? For example, consider the case where the planning tree is fully expanded (as in the absolute failure case) and absolute success has not been achieved, but there are nodes in the tree representing states that satisfy the task. In this situation we have plans in the tree with a nonzero probability of success; that is, for those plans we could specify an outcome for each operator in the plan that would cause the plan to satisfy the task. Since the search tree is fully expanded the planning program must stop, but what does it return as a result? A reasonable answer might be to return the plan with the highest probability of success, where the probability of success of a plan is defined to be the sum of the probabilities of occurrence of the plan's nodes that satisfy the task.

If we are willing to accept plans with a probability of success less than one, then perhaps we should consider stopping the planner before achieving absolute success or full expansion of the tree. For example, we might stop the planner whenever it has found a plan whose probability of success is greater than some threshold. Such a stopping criterion would have the advantage of preventing the system from expending planning effort in a situation where a plan has already been found that is almost certain to succeed.

The system's plan executor can deal with less than absolutely successful planning by being prepared to recall the planner when a terminal node of a plan is achieved and the task is still not satisfied. It may be advantageous to recall the planner before such a terminal node is reached; namely, when a node is achieved in the plan no offspring of which satisfies the task. Even though more plan steps might remain to be taken in this situation, they will not satisfy the task and the planner may determine that some other sequence of steps is more desirable to successfully complete the execution. In fact, one could argue that the planner should be given the opportunity to continue expansion of the search tree after each execution step, since the probabilities of occurrence of nodes and success of plans changes at each step; such a strategy seems an extreme one and probably cannot be justified practically.

Just as the planning program could stop with less than absolute success, there are clear cases when it should stop with less than absolute failure. Two such cases come to mind. The first is where the search tree has been expanded sufficiently to allow determination that no plan in the tree will be able to qualify as a successful plan. This would typically happen when every plan in the tree has a probability greater than some threshold of achieving a terminal node that does not satisfy the task. The second case is where the probability of each unexpanded node in the search tree is less than some small threshold. In this situation the search tree has been expanded to such an

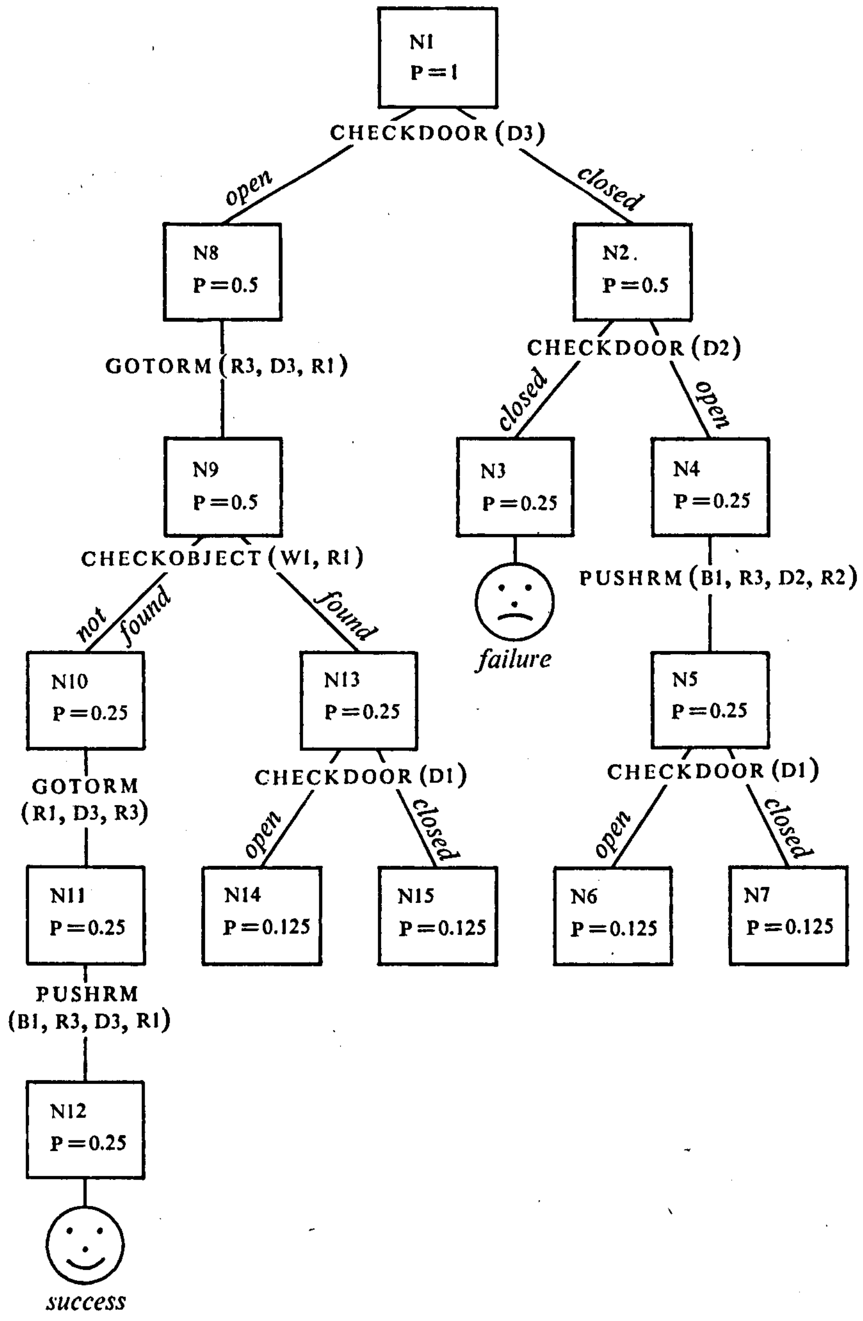


Figure 3. An example of a conditional plan.

extent that none of the states to be considered are very likely and therefore any further planning effort will only minimally increase the probability of success of any plan in the tree.

During planning, a node selection function will be needed to direct expansion of the search tree. One reasonable selection strategy is to first select the most promising plan in the tree and then select a node in that plan for expansion. Plans could be evaluated on such criteria as the probabilities of the plan's success and failure nodes, distance estimates to the goal from the plan's unexpanded nodes, and so on. Nodes within a plan could be evaluated on such criteria as estimated distance to goal and probability of occurrence.

An interesting example of the use of multiple-outcome operators can be seen by reconsidering the wedge and boxes problem discussed in Section 2 above. In the problem we wish the robot to move box B1 into room R1 under the constraint that there should never be a box in the same room with a wedge W1 (see figure 2). Assume now that the system does not know the status (open or closed) of the doors D1, D2, and D3, nor does it know if wedge W1 is in room R1. Hence, the desired plan will contain checks on the status of doors and a check to see if wedge W1 is in room R1. If we further assume that the planner halts when all unexpanded nodes have probability less than 0·2, then the plan shown in figure 3 might be the output from the planner.

This plan begins by checking the status of door D3. If D3 is closed, then D2 is checked. If D2 is closed then there are no relevant applicable operators and therefore this is a failure node. In the case where D2 is open, then box B1 is pushed into room R2 and door D1 is checked. The probability of occurrence of each of the nodes resulting from the check of D1 is less than 0·2; hence they are not expanded further. Back up at the beginning of the plan, if door D3 is found to be open, then the robot goes to room R1 and checks for wedge W1. If W1 is not found then the plan is to go back to room R3 and push box B1 into room R1; this branch of the plan completes the task and therefore forms a success node. In the case where W1 is found to be in room R1, a check is made on door D1. The probability of occurrence of each of the nodes resulting from this check is less than 0·2, and they are therefore not expanded. For this plan the probability of success is 0·25, the probability of failure is 0·25, and the remaining possible outcomes are unexpanded.

5. TEAMS OF ROBOTS

Thus far we have considered robot systems having only a single effector with which to manipulate the world. We now consider some of the problems posed by multiple robot systems.

Let us agree first that a multiple or team robot system is a system composed of several effectors controlled, at least in part, by a common set of programs. If this were not the case, then we would have several single robots whose environments happened to be complicated by the presence of other robots.

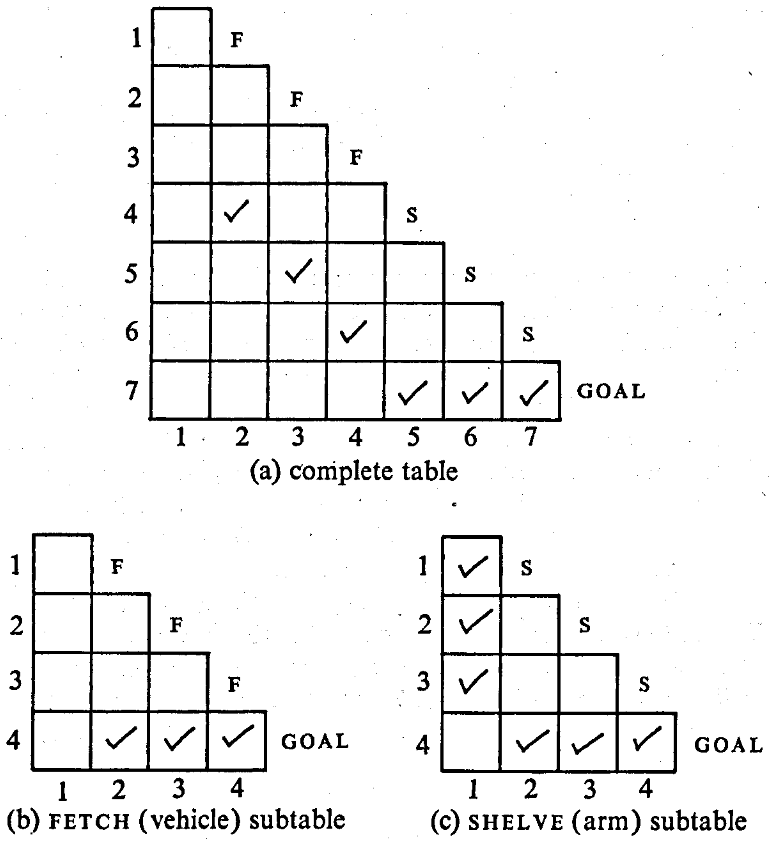
To make the discussion specific, imagine a system with two effectors: a mobile vehicle that can fetch boxes, and an arm that can lift boxes and put them on a shelf. For simplicity, we will say that these abilities are characterized by only two operators, FETCH and SHELVE. The preconditions for SHELVing a box are that the box be in the workspace of the arm. There are no preconditions for FETCHing a box (there is an inexhaustible supply of boxes and the vehicle can always bring one to the arm).

We now give the robot system the task of putting three identical boxes on the shelf. It is interesting first to note that typical current problem-solving programs would not need to recognize explicitly that the two available operators are actually implemented by separate devices. Oblivious to this fact, a successful problem solver would produce any of a number of satisfactory plans, including, for example, the sequence FETCH, FETCH, FETCH, SHELVE, SHELVE, SHELVE.

Our interest now centers on the real-world execution of this plan. In particular, we would consider a robot system a bit stupid if it waited until all three FETCHes were successfully executed before it proceeded to SHELVE. What is needed here is a means of analyzing the interdependencies of the components of the plan and of concurrently executing independent actions of the two effectors. Now, the general issue of partitioning a process into partially independent subprocesses is a complicated one that arises in studies of parallel computation (Holt and Commoner 1970). Here, though, we can make use of the triangle-table format to obtain the dependencies directly. Figure 4a is a sketch of the triangle table for the complete plan. Recall that a mark in a cell means that the operator heading the column produced an effect needed to help establish the preconditions of the operator on the same row. The bottom row of the table corresponds to the 'precondition' GOAL, while the first column of the table corresponds to preconditions provided by the initial state of the world. (Since FETCH has no preconditions, and since all of the preconditions for SHELVE are established by FETCHes, the first column in figure 4a has no marks.)

The complete triangle table can be partitioned into a pair of subtables as shown in figures 4b and 4c. To create the FETCH subtable, we copy all marks in F columns and non-F rows of the complete table into the bottom row of the subtable. To create the SHELVE subtable, we copy all marks in S rows and non-S columns of the complete table into the first column of the subtable. Notice the motivation here: the purpose of each FETCH is to satisfy a condition external to the FETCHing robot, and so are external (bottom row) goals. Similarly, the preconditions for the SHELVing robot are established by an agent external to it, and so are external (first column) preconditions. (We defer for a moment discussing a procedure for partitioning a complete triangle table when the effectors are more intimately interleaved.)

Once the two subtables have been constructed, the PLANEX algorithm described earlier can be used – in a slightly modified form – to direct the actions of the two effectors. It is plain from figure 4b that the vehicle can fetch boxes independently of any arm actions, since the arm contributes nothing toward establishing preconditions for FETCHing. The situation with respect to the arm is a little more complicated. Using the standard PLANEX algorithm, the arm could not execute a SHELVE action until all the FETCH operations had been completed, which of course is the very thing we are trying to avoid.



(a) complete table

(b) FETCH (vehicle) subtable

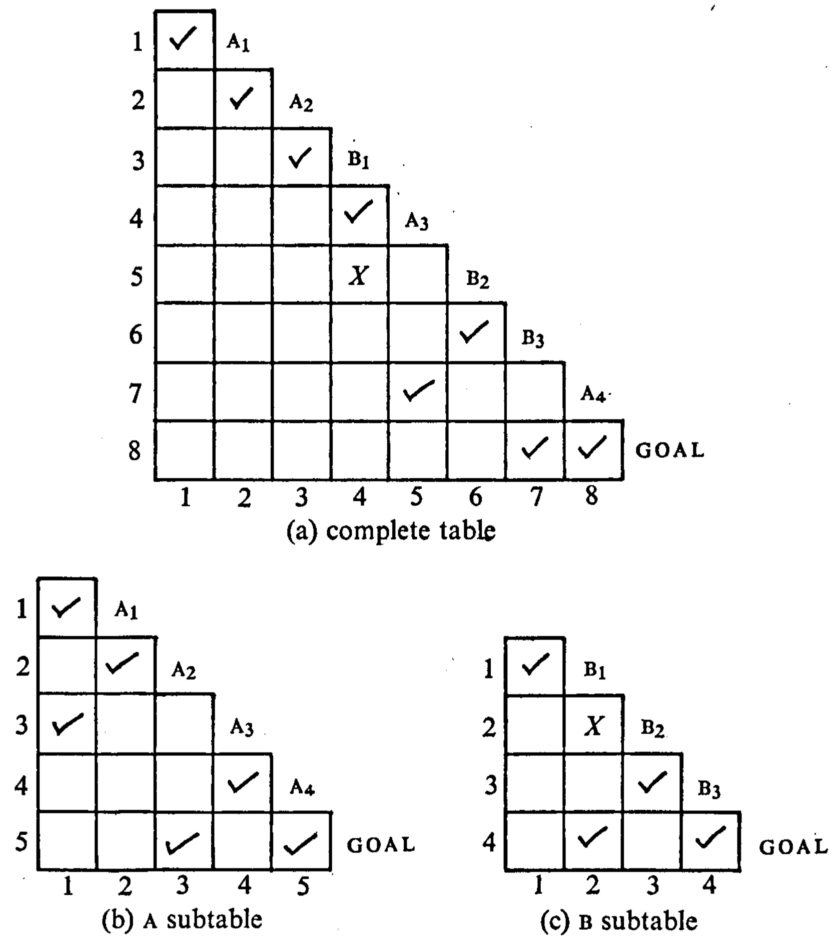
(c) SHELVE (arm) subtable

Figure 4. Triangle tables for FETCH–SHELVE example.

We must therefore distinguish between those preconditions of SHELVing that are to be established by the other effector, and those preconditions that must simply be true in the initial world (there are none of the latter in our example). Upon making this distinction, we recognize that, in order to execute the first SHELVE operation, we need only have achieved the effect marked in the top cell of the first column of figure 4c; the other effects marked in succeeding cells of that column will be added by an active agent external to the arm. In other words, whereas a single robot must have all of the basic preconditions of its plan satisfied before it begins executing the plan, the team robot can have some of the initial requirements of its plan satisfied by other robots while its own plan is being executed.

The foregoing discussion can be made more precise and, we believe, can be formulated into a straightforward algorithm for executing linear plans for multiple robots. Instead of pursuing this algorithm, let us return to the question of partitioning a complete two-robot triangle table.

We will present the algorithm by means of an example. Figure 5a shows a complete triangle table for an imaginary system composed of two robots, A and B. For present purposes we need not consider in detail the variety of different operations that A and B can perform; we need only note in the complete table which robot performs each operation. The *X* in cell (5, 4) is a mark like all the others, but merely for purposes of exposition we have distinguished between the two marks in column 4.



	1	2	3	4	5	6	7	8	
1	✓	A_1							
2		✓	A_2						
3			✓	B_1					
4				✓	A_3				
5				*X*		B_2			
6						✓	B_3		
7					✓			A_4	
8							✓	✓	GOAL

(a) complete table

	1	2	3	4	5	
1	✓	A_1				
2		✓	A_2			
3	✓			A_3		
4				✓	A_4	
5			✓		✓	GOAL

(b) A subtable

	1	2	3	4	
1	✓	B_1			
2		*X*	B_2		
3			✓	B_3	
4		✓		✓	GOAL

(c) B subtable

Figure 5. Triangle tables for algorithm illustration.

The algorithm proceeds by scanning the columns corresponding to either of the robots; let us begin with B, just to illustrate this insensitivity to order. Every mark in a B column and a non-B row is entered in the corresponding column of the bottom row of the B subtable. Cell (4, 4) of the complete table contains such a mark, and this mark is therefore entered in cell (4, 2) of the B subtable. A mark in a B column and B row of the complete table is entered into the corresponding cell of the subtable; cell (5, 4) contains such a mark. Next, we consider the B rows of the complete table. Every mark in a B row and non-B column is entered in the first column of the corresponding row of the subtable. Cell (3, 3) of the complete table contains such a mark, and this mark is entered into cell (1, 1) of the B subtable. When this process of scanning rows and columns is completed for the B robot, it is repeated for the A robot, producing finally the two subtables shown.

It appears that this basic algorithm is easily extended to any number of robots. If we had n robots, we would partition them into two subclasses: robot 1 and the remaining $n-1$. After processing this 'two-robot' system, we would recursively process the subtable corresponding to the $n-1$ system, and so forth, until n subtables were produced. However, we have not analysed whether systems of more than two robots can encounter such difficulties as 'locked states,' in which each robot waits for the results of all the others.

A final speculation concerns extending the notion of problem-partitioning from the execution phase into the planning phase. Specifically, we would like to give our multi-robot system an executive planner that has a rough characterization about the abilities of each of its effectors, and subordinate planners that do detailed planning for each effector once its goals have been established by the executive. To use the preceding example, we would like the executive planner to recognize that the initial problem can be partitioned into the two subproblems 'fetch three boxes' and 'shelve three boxes.' The ability to partition a problem in this fashion requires, of course, a deep understanding of its structure.

6. OTHER ISSUES OF INTEREST

In the foregoing sections we discussed a number of topics in robot problem-solving research that, we think, may be profitably explored. Of course, we do not mean to imply that there are no other worthy avenues of research. On the contrary, one could compile a long list of such possibilities. In the remainder of this section we mention a few of them, together with some personal observations on the issues they raise.

Task specification languages

A number of early problem-solving programs accepted problem statements in the form of a static assertion to be made true. (STRIPS, obviously, falls in this class.) We are increasingly convinced of the inadequacy of this formulation, chiefly because there is a large class of tasks that are most naturally posed in procedural terms (Fikes 1970). For example, the task 'Turn off the lights in every room on the corridor' can be posed procedurally in a straightforward manner, but can be posed assertionally only by enumerating the rooms or by introducing awkward notations.

As a general observation, we note that there is a trend for problem solvers to avoid explicit use of state variables because of the attending frame problem. But suppressing state variables makes it difficult to pose problems requiring the identification of distinguished states. Thus, for example, STRIPS cannot naturally be given the problem 'Bring the box from Room 3' because the description of the box depends on a property of the current state.

The importance of incorporating procedural information in problem solvers is now generally recognized. Our point here is that a procedural formalism is badly needed even to *state* an interestingly broad class of problems.

Multiple levels of planning

Two topics of interest entailing multiple levels of planning have arisen from our work with the SRI robot system. The first results from our learning experiments in which we store plans in memory to be used during the creation of future plans. Roughly speaking, we create a description of a plan when we store it away so that STRIPS can consider it to be a new operator, or MACROP, and can include it as a single step in a new plan. A major problem arises when one attempts to generate automatically the operator description for the plan to be stored. Specifically, the components of that description are too detailed. Typically the preconditions of a STRIPS operator will have 4 or 5 statements in it; the preconditions for a typical MACROP might have 15 or 20 such statements. If that MACROP now becomes a single step in some new MACROP, then the preconditions will be even larger in the new operator description. This explosion in precondition complexity is clearly a major barrier to the bootstrapping capabilities of this system.

The source of this difficulty is that the level of detail in a MACROP description does not match the level of planning at which the MACROP is to be used. Somehow, there must be an abstraction process to suppress detail during the construction of a MACROP description. One way in which this suppression of detail might be accomplished is by including only some subset of the preconditions and effects that would normally appear in the MACROP description. The challenge, obviously, is to determine some reasonable criteria for determining those subsets. It may be necessary to consider more sophisticated abstraction schemes involving alteration of predicate meanings or creation of new predicates. For example, the meaning of a predicate such as 'location' in human planning depends on the level at which the planning is being done. In the initial planning stages for a trip to the *MI*-7 conference it is sufficient to consider the location of the conference to be Edinburgh. When planning hotel accommodations, the location of the conference as a street address is of interest. On arrival for the first conference session, the location of the conference as a room number becomes important. Similarly, a robot planner needs to employ meanings for predicates that match its planning level.

The second interesting issue concerning multiple levels of planning is communication among the levels; in particular, transmission of information concerning failures. For example, in the current SRI robot system, STRIPS considers it sufficient for applying GOTO(BOX1) to get the robot into the same room as BOX1. When GOTO(BOX1) is executed, a lower level path-finding routine attempts to plan a path for the robot to BOX1. If no such path exists, the routine exits to PLANEX, which in most cases will try the same GOTO action again. Even if a replanning activity is initiated, STRIPS might still generate a plan involving GOTO(BOX1) under similar circumstances, and the system will continue to flounder. The problem here is the lack of communication between the path finding planner and STRIPS. Somehow, the failure of the path planner should be known to a planning level that can consider having the robot move obstacles or re-enter the room from another door.

In general, what is needed is a system capable of planning at several levels of abstraction and having the appropriate operator descriptions and world models for each planning level. Given a task, a plan is constructed at the highest level of abstraction with the least amount of detail. The planner then descends a level and attempts to form a plan at the new level. The higher-level plan is used to guide the new planning effort so that in effect the attempt is to reform the higher-level plan at the current level of detail. This process continues until the desired level of detail is obtained or a failure occurs. In the case of a failure, the result of the more detailed planning needs to include some information as to why the failure occurred, so that an intelligent alternative plan can be formulated at a higher level.

'Ad hocism' in robot planning

An important issue in the design of planners for robot systems centers on the amount of '*ad hoc*ism' one will allow to be preprogrammed into the system. STRIPS, to cite one example, is a general purpose planner requiring only simple operator descriptions and a set of axioms describing the initial state of the world. Our design of STRIPS can be viewed as an extreme on the side of generality, emphasizing automatic mechanisms for extracting planning decision criteria from any problem environment presented.

A currently popular strategy for designing a planning program is to use one of the new languages, PLANNER (Hewitt 1971) or QA4 (Derksen, Rulifson and Waldinger 1972), and write programs as consequent theorems and antecedent theorems instead of operator descriptions. This approach has the advantage of allowing the person defining the problem environment to guide the plan formation process by ordering goals and giving advice as to which operators to consider. In general, this means that the problem definer is free to build into the programs information about the problem domain that will aid in the planning process.

These alternative approaches prompt an inquiry into the goals of problem-solving research. If the purpose is to design a planner for a fixed physical environment, a given robot with a fixed set of action routines, and a fixed set of predicates and functions to appear in the models, then one builds into the system as much specific information as possible about how to accomplish tasks in the environment. For example, one might specify the optimal paths between any two rooms, provide recognition programs for each object in the environment, and so forth. Such a system might be useful for some particular application, but interest in its design as a research task is minimal.

What kind of robot planning system is of interest for problem-solving research? We suggest that a key issue in the consideration of that question is the system's capability of being extended. One type of extendability of interest is exploration. That is, we want a robot system to be capable of going into new areas of its environment, adding information obtained in the exploration to its world model, and finally, performing tasks in the new area. An exploration capability limits the level of '*ad hoc*ness' we can program into the system. For example, the planner cannot depend on preprogrammed paths between rooms and preprogrammed object recognition programs if exploration is going to bring new rooms and new objects into consideration.

A second type of extendability of interest to us is the learning of new action capabilities. One would like the system to produce automatically new action routines (like, for example, the STRIPS MACROPS) from its experience; but even merely allowing human introduction of additional action programs places restrictions on the level of '*ad hoc*ness' in the system. For example, if there is only one action routine in the system for moving a box from one room to some other room, and if we know that no new actions will ever be added to the system, then we could preprogram the planner to call that action whenever a box was needed in a room. But if we allow the possibility of a new action being added to the system that, say, is specially designed to move a box to an *adjacent* room, then we want our system design to be such that the new program can be easily added and that the planner will then make intelligent use of it.

Another aspect of the extendability issue is that a system should be a good experimental tool for research. That is, we might want to be continually extending the system to include new areas of interest such as those discussed in this paper. Hence, we would like the basic formalism and structure of our system to be flexible enough to allow exploration of these areas without requiring large amounts of reprogramming effort.

In conclusion, then, we can say that it seems certainly worthwhile to provide facilities in planning programs for easy specification of *ad hoc* information, and STRIPS is still deficient in that regard. The danger seems to be that AI researchers may be seduced into designing systems that are so dependent on advice about specific situations that they have no extendability and are little more than programs containing solutions to a very restricted set of problems. Our discussion in this section has attempted to point out that one way to temper the nature and extent of a problem-solving program's dependence on *ad hoc* information is to consider various ways in which one might wish the program to be extendable.

Acknowledgement

This research was sponsored by the Office of Naval Research under Contract N00014-71-C-0294 and the Defence Advanced Research Projects Agency under Contract DAHC04-72-C-0008.

REFERENCES

Bruce, B.C. (1972) A model for temporal references and its application in a question-answering program. *Art. Int.*, **3**, 1-25.

Derksen, J., Rulifson, J.F. & Waldinger, R.J. (1972) QA4 language applied to robot planning. *AIC Technical Note 65*, SRI Project 8721. California: Stanford Research Institute.

Fikes, R.E. (1970) REF-ARF: a system for solving problems stated as procedures. *Art. Int.*, **1**, 27-120.

Fikes, R.E. & Nilsson, N.J. (1971) STRIPS: a new approach to the application of theorem proving to problem solving. *Art. Int.*, **2**, 189-208.

Fikes, R.E., Hart, P.E. & Nilsson, N.J. (in press) Learning and executing robot plans. *Art. Int.*

Green, C.C. (1969) Application of theorem proving to problem solving. *Proc. Int. Joint Conf. on Art. Int.*, pp. 219-40. Washington DC.

Hart, P.E., Nilsson, N.J. & Robinson, A.E. (1971) A causality representation for enriched robot task domains. Tech. Report for Office of Naval Research, Contract N00014-71-C-0294, SRI Project 1187. California: Stanford Research Institute.

Hewitt, C. (1971) Procedural embedding of knowledge in PLANNER. *Proc. Second Int. Joint Conf. Art. Int.*, pp. 167-82. London: British Computer Society.

Holt, A. & Commoner, F. (1970) Events and conditions. *Record of Project MAC Conf. Concurrent Systems and Parallel Computation*, pp. 1-52. Massachusetts: Association for Computing Machinery.

Munson, J.H. (1971) Robot planning, execution, and monitoring in an uncertain environment. *AIC Technical Note 59*, SRI Project 8259. California: Stanford Research Institute.

Winograd, T. (1971) Procedures as a representation for data in a computer program for understanding natural language. *A.I. Technical Report 17*, MIT. Artificial Intelligence Laboratory, Cambridge, Mass. MIT. Also available as *Understanding Natural Language.* Edinburgh: Edinburgh University Press 1972.

Yates, R., private communication.