April 1973

# A HIERARCHICAL ROBOT PLANNING AND EXECUTION SYSTEM

by

Nils J. Nilsson

Artificial Intelligence Center

Technical Note 76

SRI Project 1187

The research reported herein was sponsored by the Office of Naval Research under Contract N00014-71-C-0294. This work was done in conjunction with another project supported in part by the Advanced Research Projects Agency through Contract DAHC04-72-C-0008.

ABSTRACT

This report describes a robot control program consisting of a hierarchically organized plan generation and execution system. The program is written in QA4 and makes use of several features of that language. The usually sharp distinction between robot plan generation and execution is intentionally blurred in this system in that planning and execution phases occur intermixed at various levels of the hierarchy. The system currently exists as a running program that clearly illustrates the concepts involved; major additions and refinements would be necessary if the system were to be used to control an actual robot device.

CONTENTS



ILLUSTRATIONS

# I INTRODUCTION

This report describes a model system that generates and executes robot plans. There have been several systems of programs that control robot hardware (see, for example, Paul[1]*); these typically have had limited, if any, general planning ability. Similarly, there have been several systems that generate robot plans (see, for example, Sussman[2], Winograd[3], and Derkson, et al.[4]), but these were not connected to systems that actually executed these plans. The major instance of a system that both generated and executed plans was the STRIPS-PLANEX system[5,6] that controlled the SRI mobile robot SHAKEY.

Several important issues arise in the design of planning-execution systems for robots. The STRIPS-PLANEX system faced some of these, most notably those having to do with intelligent monitoring of the plan as it is executed. Several additional problems not adequately handled by STRIPS-PLANEX are described in some detail in Reference 7. In this study, we have focused in particular on the problem of hierarchical planning and execution. What we desire is a system that can generate plans at various levels of detail and monitor the execution of these plans in a manner well integrated with the planner.

---

* References are listed at the end of this report.

The system we will be describing has the following general characteristics. When given a task, it generates a plan to perform this task. The plan is in terms of rather high-level actions, and typically the execution of any of these actions will involve the generation of a plan at a slightly more detailed level. The execution of the more detailed plan may involve the generation of a still more detailed plan, and so on, recursively down a hierarchy of detail levels.

Such a strategy for plan generation has the obvious advantage of limiting the search for plans at each level to short plans (say those with no more than about five steps.) Perhaps less obviously, a hierarchical plan-generating system seems to be required if the system is to learn automatically new "macro" actions composed of a sequence of more basic actions. The problems involved in learning, generating, and executing plans composed of macro-actions depend for their solution on the development of an appropriate hierarchical system. Furthermore, we should seek a certain uniformity of planning and execution strategy at each level of the system in order to simplify the task of adding new levels to the top.

Some of these same issues have been faced in other hierarchical systems such as ABSTRIPS by Sacerdoti[8] and LAWALY by Siklossy and Dreussi[9]. The present system differs somewhat from these, both in the way plans are generated and in the way they are executed and monitored.

Our system is designed to work on absolutely trivial robot problems rather than on difficult ones, and a word or two seems appropriate about this choice. At the moment we are not concerned with a "performance system";

rather we are concerned about understanding with some clarity the fundamental issues involved in building a hierarchical system. Admittedly our design might have to be changed drastically before it will solve more difficult robot problems, but we regard the development of a model system to be a worthwhile first step. Too often when one approaches a large problem with the "let's-just-program-it-up" attitude, one gets very little understanding of why the system works as it does. We take it to be a primary goal of this work to develop a methodology or at least a point of view about how such robot systems ought to be organized. The best way to explicate our present point of view is to describe a simple model.

Our system operates in a simple simulated environment. (It would probably not be too difficult to have it actually control the SHAKEY hardware, but for simplicity we have decided, for the present, to simulate execution.) The program is written in QA4[10] and makes use of several of the features of that language. In particular, we rely heavily on pattern-directed function invocation; automatic backtracking; processes; retrieval and storage of expressions in the QA4 net; associational property lists with expressions; and demons. We assume the reader has some familiarity with these ideas.

The world for our robot is a simple, infinite x-y coordinate system. Besides the robot, there exists an immovable BOX that is an obstacle to navigation. There may be other, pushable objects in the world; for example an object named THING is one such. The primitive actions of the robot are: rolling forward a certain number of units and turning (rotating) a certain

number of degrees in either direction. Combining these, it can travel around its world and it can push objects. Even with such a trivial world, we can design and test quite complex hierarchical systems.

Our robot system models its world in a structure we shall call the WORLD-MODEL. (In QA4, we store assertions about the WORLD-MODEL in the ETERNITY context.) The WORLD-MODEL contains assertions about the location of various objects in the world. For example, the WORLD-MODEL to be used in a later example contains the following assertions (shown later in Figure 2):

```
(AT ROBOT      0   0)
(HEADING ROBOT     0)
(AT BOX        7   1)
(AT THING     -4 -12)
```

Assertions in the WORLD-MODEL are customarily made or deleted by those robot actions that simulate moving in the world. (They are not subject to removal by QA4 backtracking, for example.)

When the system is generating a plan, it does so with reference to a PLANNING-MODEL. When it is set up, the PLANNING-MODEL is a copy of the present WORLD-MODEL. (In QA4, we create and use an immediate descendant of the GLOBAL context for this.) All proposed actions are allowed to have their effect on the PLANNING-MODEL. These effects can be backtracked if an alternative plan needs to be considered.

Our description of the system will consist of two main parts. First we shall describe how the system works in the ideal case, that is in a case uncomplicated by failures in planning, surprises in the world, or errors of execution.

Second we shall describe what we have done to deal with failures and surprises.

The system currently exists as a running program although some of the error-handling mechanisms are still incomplete and awaiting further refinements to the QA4 language.

## II SYSTEM DESCRIPTION: MAJOR COMPONENTS

### A. ACTIONS

The fundamental building unit of our hierarchical robot system is the ACTION. Each ACTION is a piece of program, together with its arguments; when it is run it produces some or all of three important effects. The first is simply an effect on the world, such as a robot motion. Secondly, an ACTION may produce or add to a PLAN. A PLAN is an ordered list of ACTIONS. An ACTION may call another ACTION as a subroutine. In such a case the subordinate ACTION may add components to some PLAN that the main ACTION is producing. Thirdly, an ACTION may make changes to various MODELS of the world that the robot system maintains. If the ACTION has an effect on the world, it records this effect in the WORLD-MODEL. If the ACTION produces a PLAN, the predicted effects of running this plan are recorded in a PLANNING-MODEL that has been specially set up prior to running the ACTION routine.

The interesting thing about the ACTION programs is that they combine both planning and execution functions in one program structure and thus blur the usual distinction between planning and execution. Some ACTIONs may produce PLANs but have no effects on the world. Others may produce only effects on the world but no PLANs. ACTIONs might produce PLANs and effects on the world. Indeed the same ACTION might in some situations produce PLANs but no

world effects and in other situations just the opposite! In any case all planning and execution are diffused throughout the entire system in such ACTION programs; there is no special top-level planning program that produces plans to be executed by a top level executive as was the case in STRIPS-PLANEX.

Discussing some examples will give a clearer understanding of the role of ACTION programs. For the simple robot world we discussed earlier, we have so far implemented the following ACTION programs:

| | |
|---|---|
| GOTHERE | PUSHTHERE |
| GOTO | PUSHTO |
| ROLLTO | PUSH |
| POINTAT | PUSHING |

These can be described as follows:

(1) GOTHERE is a QA4 program that takes as an argument the pattern (AT ROBOT GX GY). It can be called by pattern-directed invocation; that is, it is subject to call whenever the QA4 interpreter meets a GOAL statement of the form (GOAL (AT ROBOT GX GY)). (Presently GOTHERE is the only ACTION that we have implemented that would be called by this GOAL statement, but in principle we could have others also. The QA4 backtracking system would then ultimately select the appropriate ACTION.) When GOTHERE is invoked, it checks a PLANNING-MODEL to see if the (immovable) BOX overlaps the goal location, GX GY. If the BOX obstructs the goal, GOTHERE fails. (We will discuss the implications of such failures later.)

Otherwise GOTHERE adds the ACTION GOTO(GX GY) to whatever PLAN is being constructed at the time. GOTHERE itself produces no effect on the world or WORLD-MODEL, but it does update a PLANNING-MODEL.

(2) GOTO(GX GY) checks a PLANNING-MODEL to see if there is a clear path (avoiding BOX) from the robot's position to (GX GY). If there is, it adds ROLLTO(GX GY) to the PLAN that it is constructing. Otherwise it computes a subgoal location, (GGX GGY) well removed from BOX and executes the pair of GOAL statements: (GOAL (AT ROBOT GGX GGY)), GOAL (AT ROBOT GX GY)). When these GOAL statements are executed, any effects they might have on a MODEL are effects on the PLANNING-MODEL other than on the WORLD-MODEL. The GOAL statements invoke calls upon GOTHERE, and the end result is to produce the PLAN {GOTO(GGX GGY), GOTO(GX GY)}.

(3) ROLLTO(GX GY) first computes the angular direction from the robot's position to (GX GY). It calls the ACTION POINTAT to point the robot in that direction. It then computes the distance the robot must roll to reach (GX GY) and causes the robot to roll that far. If, after this, the robot does not think it is sufficiently close to (GX GY), ROLLTO fails. (We will discuss this type of failure later.) ROLLTO and POINTAT do not generate PLANs but they do update the robot's WORLD-MODEL with its new location and orientation.

(4) PUSHTHERE acts in a manner parallel to GOTHERE. It too can be called by pattern-directed invocation. Its argument is the pattern (AT OBJECT GX GY), so it can be called whenever the QA4 interpreter meets a GOAL statement of the form (GOAL (AT OBJECT GX GY)). (PUSHTHERE is the only ACTION implemented so far that can be called by this GOAL statement.)

When PUSHTHERE is invoked, it first checks to see if the OBJECT is either the ROBOT or the BOX. If it is either, it fails. Next it checks the PLANNING-MODEL to see if the BOX overlaps the goal; if it does, PUSHTHERE fails. Otherwise PUSHTHERE adds the ACTION PUSHTO(OBJECT GX GY) to whatever PLAN is being constructed at the time. PUSHTHERE itself produces no effects on the world or WORLD-MODEL. It updates a PLANNING-MODEL.

(5) PUSHTO(OBJECT GX GY) checks a PLANNING-MODEL to see if there is a clear path (avoiding BOX) from the OBJECT's position to the goal (GX GY). If there is, it adds PUSH(OBJECT GX GY) to the PLAN. Otherwise it computes a subgoal location (GGX GGY) and uses the GOAL mechanism to compute a PLAN for moving OBJECT first to the subgoal and then to the goal (avoiding the BOX). The PLAN is computed by calls to PUSHTHERE and would presently consist of {PUSHTO(OBJECT GGX GGY), PUSHTO(OBJECT GX GY)}. PUSHTO updates a PLANNING-MODEL.

(6) PUSH(OBJECT GX GY) calculates a place (PX PY) to which the robot should move to begin pushing the object. Through a pattern-directed call to GOTHERE, it then adds GOTO(PX PY) to the PLAN being constructed. Next it calculates a place (X Y) to which the robot should roll so that the OBJECT will result in being pushed to (GX GY). Then it adds ROLLTO(X Y) and PUSHING(OBJECT GX GY) to the PLAN.

(7) PUSHING(OBJECT GX GY) merely updates the robot's WORLD-MODEL with the OBJECT's new location. It neither generates a PLAN nor has any effect on the world.

We note that all of the planning is done in a hierarchical fashion under the control of the various ACTION programs. Through the use of the QA4 GOAL statement, planning that requires "search" can also be done by the system. QA4 takes care of such search processes automatically through its backtracking mechanism. In our example to be presented later, search does not play a role because we have only written one ACTION program (PUSHTHERE and GOTHERE) for each of the two major GOAL forms that can be handled. But one can easily imagine a larger system that could respond to several different kinds of GOAL statements each matching the patterns of several different ACTION programs.

Our approach involves hierarchies of ACTION programs--in our case we have the GO hierarchy and the PUSH hierarchy. At the top of each hierarchy is an action that can be invoked through a GOAL statement. Thus at any

level of the system whenever a GOAL statement is used, the resulting PLAN is in terms of top level ACTIONS.

We note that GOAL statements themselves can be thought of as ACTION programs. When they are run, some other ACTION program is invoked that might affect the external world or generate a PLAN.

B. EXECUTIVEs

EXECUTIVEs are programs that run PLANs. We run an EXECUTIVE as a QA4 process that creates offspring EXECUTIVE processes.* At any given time, there may be several EXECUTIVE processes set up and waiting to run.

The argument of an EXECUTIVE program is a PLAN. Recall that a PLAN is an ordered list of ACTIONs. For example {ROLLTO(3 4), PUSHTO(THING -2 7), GOTO(8 5), PUSHTO(THING 3 4)} is a PLAN. The EXECUTIVE program can be explained simply as an ordered application of a program called PERFORM to each ACTION in the PLAN. The simple flow chart in Figure 1 shows how PERFORM works.

C. An Example

To get a clear understanding of how the EXECUTIVE and ACTION programs interact and of how hierarchical PLANS are composed, let us consider an example taken from our simple robot world. Suppose the world and WORLD-MODEL are as shown in Figure 2. We set up a QA4 process based on the EXECUTIVE program.

---

* The QA4 process mechanism was used here mainly because it simplified the implementation of our scheme for execution monitoring to be described later.

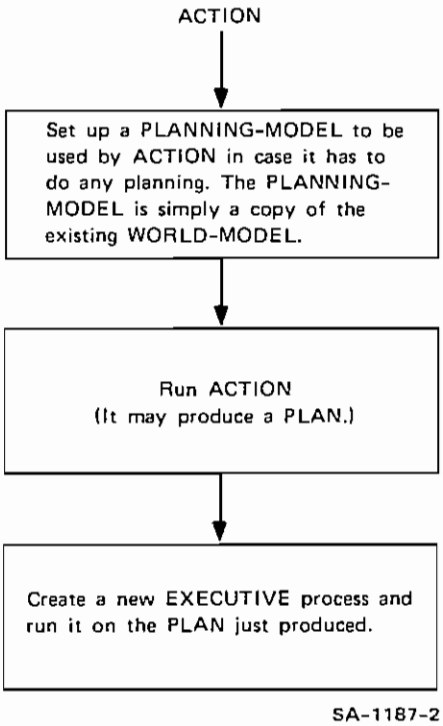


FIGURE 1 SIMPLIFIED FLOW CHART FOR PERFORM

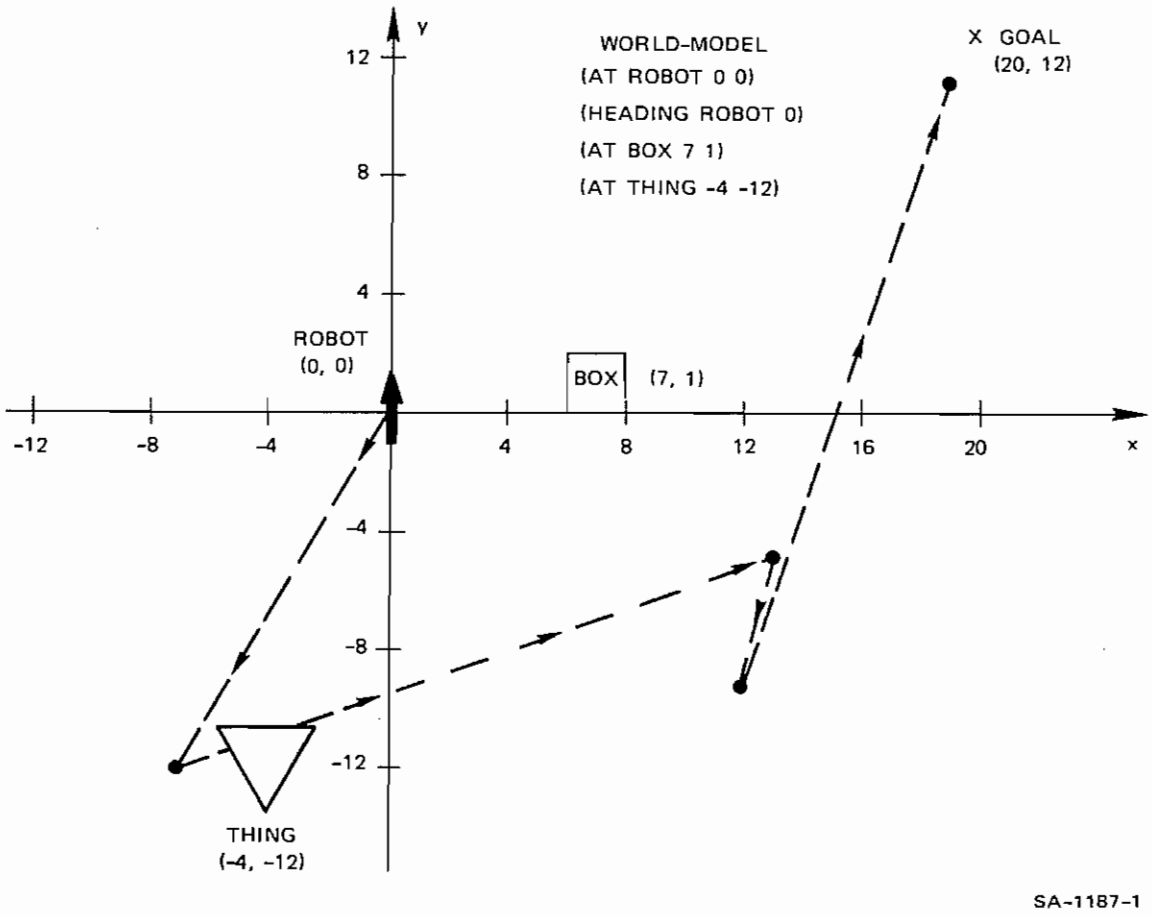


FIGURE 2 INITIAL WORLD AND WORLD-MODEL FOR EXAMPLE PROBLEM

Let the input to this process be the singleton PLAN {GOAL (AT THING 20 12)}. A trace of this process goes as follows:

(1) Since the PLAN is a singleton, the process consists merely of applying PERFORM to the ACTION (GOAL (AT THING 20 12)). A PLANNING-MODEL is set up and the ACTION is run. PUSHTHERE is invoked resulting in the creation of the PLAN {PUSHTO(THING 20 12)}. The planning model is updated, but this will have no consequences in subsequent activity.

(2) An EXECUTIVE process is set up and it is run on the new PLAN.

(3) Since the new PLAN is also a singleton, the process consists merely of applying PERFORM to the ACTION PUSHTO(THING 20 12). A new PLANNING-MODEL is set up and the ACTION is run. Since BOX is in the way of a straight path from THING to the goal, a subgoal at (14 -6) is calculated. Finally, the PLAN {PUSHTO(THING 14 -6), PUSHTO(THING 20 12)} is created through pattern-directed calls to PUSHTHERE.

(4) An EXECUTIVE process is set up and it is run on this PLAN. The creation of PLANs and EXECUTIVE processes continues like this until finally a ROLLTO is executed, taking the ROBOT near THING to push it to the designated subgoal. We tabulate the hierarchy of ACTIONS that are planned and run in Figure 3. The circled numbers indicate the order in which each ACTION is run. The path taken by the robot is shown by a dashed line in Figure 2.

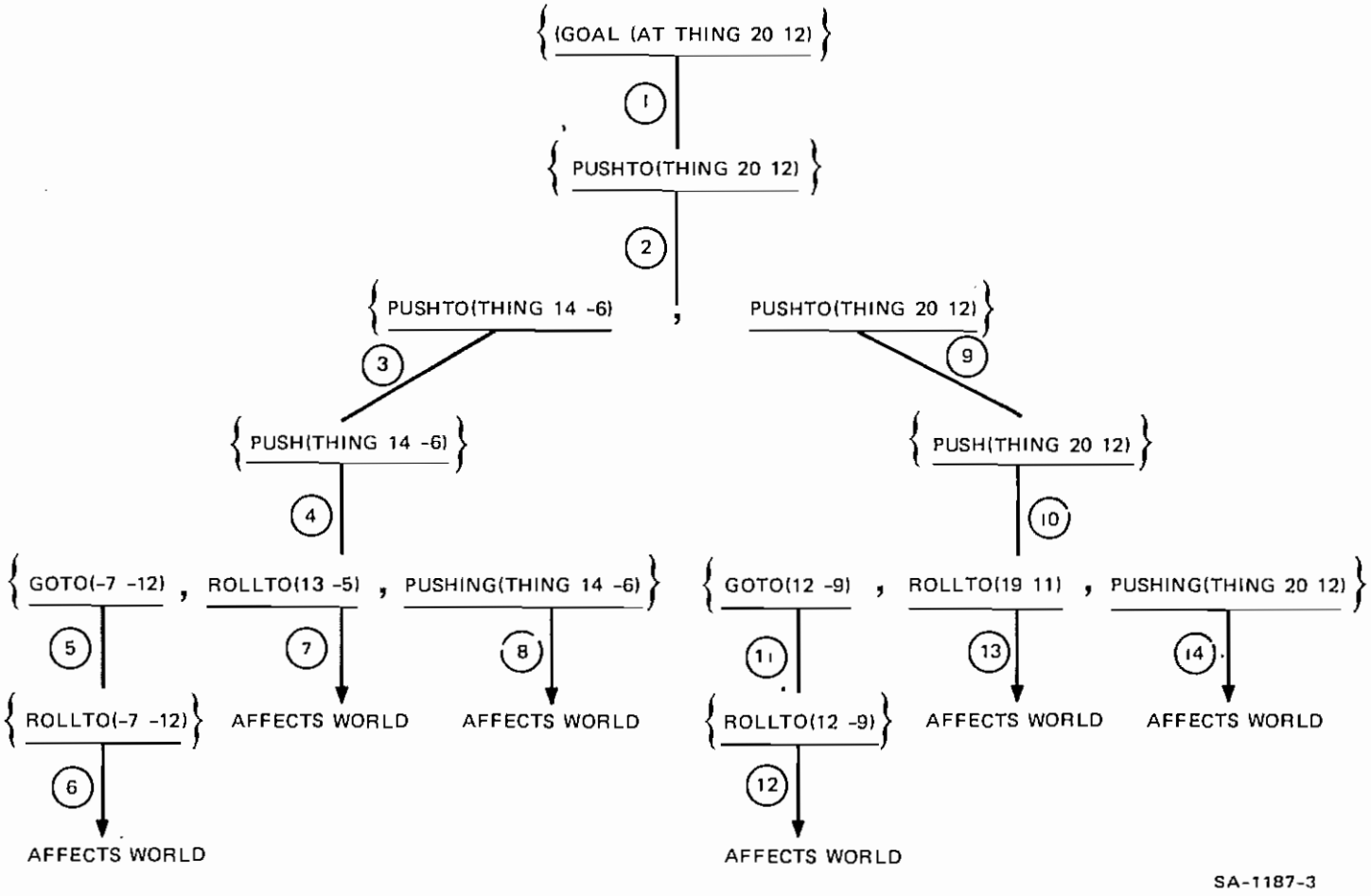


FIGURE 3 HIERARCHY OF ACTIONS RUN IN THE EXAMPLE

We see that the various levels of the PLAN are expanded in a depth-first manner. This convention may be too simplistic for more sophisticated problems, but it does afford us a good beginning strategy. Modifications might include the uniform expansion of each step in a plan down to some level at which we would dare to execute some of the steps (see Sacerdoti[8]). Our view in this study is that the choice of an expansion strategy is a second-order matter.

## III SYSTEM DESCRIPTION: FAILURES AND SURPRISES

### A. Failures

In a dynamic and inaccurately modeled world, several pitfalls prevent the smooth generation and execution of plans. For this reason, the execution of plans must be well integrated with the generation of plans, and provision must be made to compare the expected outcomes of ACTIONS with the actual outcomes. For convenience we have somewhat arbitrarily divided the difficulties that might arise into two broad classes: failures and surprises.

A failure occurs whenever a QA4 program FAILS. In QA4 such a failure invokes the backtracking mechanism that undoes all of the reversible effects of the computation occurring since the last decision point. In our use of this mechanism, the decision points would typically be at the points where GOAL statements invoke programs that attempt to satisfy the goal. If, after such a choice, planning stops due to a QA4 failure (caused for example by exhausting choices at a lower level), this backtracking resets the system to its state at the choice point and another choice is made. If ultimately a choice resulting in a viable plan is made, the system continues just as though no failure had occurred. If no plan is found at the level in question, a failure at the next highest level is generated,

and so on, until some level is reached that can adequately deal with the failure. In case no level can set the failure right, the system responds with a FAIL COMPLETELY. We use the QA4 failure mechanism quite liberally in the design of the ACTION programs. It is considered good programming style to put checks at the beginning of a program that might rule out the use of that program so that it would be eliminated from consideration sooner rather than later. (Note that in PUSHTHERE if BOX overlaps the goal location, PUSHTHERE fails. In that case there would either exist some other program that could be invoked to deal with such a difficulty or the failure would propagate upward.) The failure mechanism is a convenient way to handle automatically the various failures in planning in a hierarchical system.

Additionally, we have used the failure mechanism to signal when an ACTION program did not accomplish what was intended for it.* This use has been with the primitive actions, ROLLTO and POINTAT, in which checks are made of the ROBOT's position and heading, respectively, before exiting. If the error is not sufficiently small, a QA4 failure is generated. We have put in a test in the EXECUTIVE program to trap failures originating from primitive ACTIONs. This trap merely tries to run the failed primitive program once more (sometimes this works!). We take a second failure as a sign

---

* In our present system, in which the effects of ACTIONs on the world are simulated, we have not exercised this feature. It will be important, however, in the control of actual robots.

that something more serious is wrong, and propagate the failure up the line to be trapped by the next higher level EXECUTIVE and so on.

It appears that this application of the failure mechanism will be useful but we have not experimented with it very much yet. Larger scale experiments, perhaps with the SHAKEY hardware will be needed to gain a better appreciation of its advantages and disadvantages.

B. Surprises

In a dynamic and unknown world, assertions might dynamically appear in and disappear from the WORLD-MODEL. In the WORLD-MODEL of a robot wandering around an office environment populated by people, doors that were closed may suddenly become open and vice versa, the locations of objects and people may change, and so on--all independently of the robot's own actions. New information may appear in the WORLD-MODEL as a side-effect of the robot's actions. For example, when a robot uses vision to update its location in the model, it may "notice" a new object of which it has no previous record. Also people may make assertions in the robot's WORLD-MODEL while it is executing a task. (Many actual changes in the world will not immediately, or ever, be recorded in the WORLD-MODEL. Obviously the robot cannot be troubled by these until they do get recorded.)

Now many of these changes in the WORLD-MODEL will have no consequence for the ultimate execution of a task. They will be irrelevant happenings. We are interested here in those that could affect the way a robot executes the rest of its task. These we shall call surprises.

First, we must have a way of dealing with the happy kind of surprise that signals that some goal is satisfied before the robot expected it would be. In this case, the system should abandon all work toward making that goal true and get on with the rest of the task.

Second, there is the sort of surprise that signals that some subsequent step in the PLAN being executed will not be able to be run, even though no failure has yet occurred. Such a difficulty can be absolutely prevented only by continuous checking of the assumptions on which the PLAN is based.

In the STRIPS-PLANEX system both types of surprises were handled by the PLANEX scanning algorithm. Our view here is that this scanning was overly expensive, and that what is needed are mechanisms that achieve most of the PLANEX results with much less effort. Currently we have partially implemented a simple technique for dealing with the happy surprises. The technique involves the use of the QA4 demon mechanism.

A QA4 demon is a technique for interrupting a running program or process and transferring control temporarily to some other program or process. A demon specifies a "watch-for" condition and a program to be run whenever the condition being watched is met. In our system we store on the property list of a PLAN that particular assertion that this PLAN is supposed to achieve. Then just before an EXECUTIVE process is created to run the PLAN, we set up a demon to watch the WORLD-MODEL for the occurrence of the assertion. Our idea is that whenever this assertion occurs either as a direct result of the robot's efforts or through serendipity, the demon should interrupt whichever process is running and transfer control to the EXECUTIVE process that originally

set up the demon. In this way, the next step in the next-higher level PLAN will be executed.

At the moment, QA4 has no easy way for a demon to transfer control to the process that created it, so we are awaiting an improvement here before this use of demons can be fully tested. Our current system merely has the demon transfer control to the parent of that process running at the time the demon fires. Also there is no way yet to kill demons after they have performed their intended service, but future versions of QA4 ought to have this feature.

We have not yet implemented a technique to deal with the second kind of surprise, the type that signals that some future step in a PLAN will be unexecutable. As an example, consider a PLAN that has as one of its steps going through (an assumed open) doorway. Suppose while executing some preliminary steps in the PLAN, the robot learns that the doorway is closed. Since going through the doorway can no longer be executed, the robot might better halt the entire PLAN as soon as it learns this new fact instead of waiting for the inevitable failure when it attempts the unexecutable action.

Our feeling is that QA4 demons can be used to monitor these kinds of surprises also, although we have not yet included this feature in the program. The conditions to be watched for can be stored in the PLAN's property list, and the EXECUTIVE that creates an EXECUTIVE to run the PLAN can set up the appropriate demons. When any of the demons are activated, a QA4 failure could be generated, which would then be treated the same as any other failure.

We note that being able to deal with surprises merely improves the efficiency of the execution. In the case of happy surprises, we have achieved a goal before we expected to; if the system fails to notice this, it would presumably go on working and ultimately either notice it had already achieved the goal or achieve it again. In the case of the other surprises, the system would ultimately attempt to execute the unexecutable ACTION and a failure would be generated. Appropriate processing of surprises provides an advance notice of success or failure.*

---

*The current system lacks one major ability of STRIPS-PLANEX. The PLANEX scan algorithm would automatically recognize when early steps in a PLAN could be discarded because their effects only provided already-satisfied preconditions of later steps. This feature appears to be important, and we will devote some future effort to see how it can be incorporated into the system.

# IV DISCUSSION

This report has described a simple model system for hierarchical robot planning and execution. We presented a short example to illustrate the operation of the system. Several features of the system were not fully exercised in this example, but it is felt that these features will be important components of an expanded system designed to deal with more complex robot tasks. In particular, we might stress the importance of the following items:

(1) Hierarchically Organized ACTION Routines--Hierarchies of action routines have been important in other robot execution systems, for example the LLAs and ILAs of the SRI SHAKEY system.[11] There has been little attention to their use, however, in conjunction with a general planning system such as the QA4 interpreter. We think that any large robot system must have some sort of hierarchical organization of its actions. Our example in this report did not fully illustrate the economies in plan generation made possible by a hierarchical system, but a slightly more complex example would (if it involved several pattern-invocable ACTIONS for each GOAL statement).

(2) Uniformity of Planning and Execution Strategies--We have already mentioned that we feel a uniform structure throughout the hierarchy

will be important when we begin to consider systems that can automatically add new levels to the top of the hierarchy.

(3) A Generalized Planning Ability--Obviously each ACTION routine could have its own specialized planning or problem-solving system incorporated within it and hand-tailored to deal with the kinds of problems it encounters. However we feel that a common planning system (such as the QA4 interpreter with backtracking) is adequate and useful for many planning tasks, and thus its use allows substantial design economies for perhaps only a small cost in efficiency.

(4) Execution Monitoring--Any robot system must have a means to monitor performance. The approach taken here has relied on demons, and it is our feeling that they will play a useful role in larger systems.

It is intended that work on this system will continue; ultimately we would like to test it out on the SHAKEY hardware or its successor. Such testing will probably await the completion of a more efficient QA4 system now being developed.

ACKNOWLEDGMENT

This work was supported by the Information Systems Branch of the Office of Naval Research under Contract No. N00014-71-C-0294. This work was done in conjunction with another project supported in part by the Advanced Research Projects Agency through Contract DAHCO DAHC04-72-C-0008.

The author wishes to thank Peter Hart, Earl Sacerdoti, and Richard Fikes for their several helpful suggestions and criticisms.

REFERENCES


1. R. Paul, "Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm," Ph.D. Dissertation, Computer Science Department, Stanford University, Stanford, California (1973).

2. G. Sussman, "Teaching of Procedures-Progress Report," Artificial Intelligence Laboratory Memo No. 270, Massachusetts Institute of Technology, Cambridge, Massachusetts (October 1972).

3. T. A. Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," Ph.D. Thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts (1971).

4. J. A. Derksen, J. F. Rulifson, and R. J. Waldinger, "The QA4 Language Applied to Robot Planning," AFIPS Conference Proceedings, Vol. 41, Part II, pp. 1181-1192, Fall Joint Computer Conference (AFIPS Press, Montvale, New Jersey, 1972).

5. R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, No. 3/4, pp. 189-208 (Winter 1971).

6. R. E. Fikes, P. E. Hart and N. J. Nilsson, "Learning and Executing Generalized Robot Plans," Artificial Intelligence, Vol. 3, No. 4, pp. 251-288 (Winter 1972).

7. R. E. Fikes, P. E. Hart, and N. J. Nilsson, "New Directions in Robot Problem Solving," Machine Intelligence 7, D. Michie and B. Meltzer (editors), (Edinburgh University Press, Edinburgh, Scotland, 1972).

8. E. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Preprints of International Joint Conference on Artificial Intelligence-1973, Stanford, California (August 1973).

9. L. Siklossy and J. Dreussi, "A Hierarchy-Driven Robot Planner Which Generates Its Own Procedures," Preprints of Internation Joint Conference on Artificial Intelligence-1973, Stanford, California (August 1973).

10. J. F. Rulifson, "QA4: A Procedural Calculus for Intuitive Reasoning," Ph.D. Dissertation, Computer Science Department, Stanford University, Stanford, California (November 1972).

11. P. E. Hart, et al., "Artificial Intelligence-Research and Applications," Annual Technical Report, Contract DAH-C04-72-C-0008, SRI Project 1530, Stanford Research Institute, Menlo Park, California (December 1972).