SAND85–0958
Unlimited Release
Printed December 1985

Distribution
Category UC–38

Task Planning for Control of a Sensor-Based Robot

R. Charleene Lennox* and George F. Luger
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Raymond W. Harrigan
Intelligent Machine Systems Division
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

This paper is divided into four major sections. The first is an introduction to the research area. The second provides a brief overview of the history of "planning" in artificial intelligence (AI), while the third part of the paper describes the representational power of very-high-level computer languages and how this power may be brought to planning in robotics research. Finally, we describe our PROLOG-driven Planner for controlling a PUMA arm used for object manipulation and note the future directions of this work.

*Present address: Digital Subsystems Division I
Sandia National Laboratories
Albuquerque, NM 87185

Acknowledgments

We gratefully acknowledge Greg Starr's assistance with the PUMA-560 robot and the VAL-II robot control language. Greg Donohoe provided invaluable help in the area of computer hardware. Greg Donohoe and Mike Daily were responsible for the vision-sensing subsystem. Alan Christiansen's paper (which has been submitted for publication), "The History of Planning Methodology: An Annotated Bibliography," was especially helpful.

Task Planning for Control of a Sensor-Based Robot

Introduction

Intelligent machine systems will necessarily require the integration of sensors and artificial intelligence (AI) techniques with hierarchical manipulator control to provide a system capable of semiautonomous operation. Current robot systems have very limited sensing capability and achieve positioning through open-loop control techniques. Such systems have serious limitations in that there is no ability to adapt to unstructured or changing environments. This is a particularly severe limitation in hazardous environments where an inappropriate action can result in acute danger to personnel and equipment.

A desired goal in robotics technology is the development of sensor-based control, in which a general description of a task is taken by a machine controller, and a detailed plan is formulated internally and executed to accomplish the task. The key advantage of such robot systems would be to put the human operator in a supervisory role, making the robot system more efficient and semiautonomous in a wide variety of application areas.

Semiautonomous robot systems controlled at the task level must be highly integrated, sensor-rich systems. To the extent that the human is removed from the control loop, the capabilities of sight, touch, reasoning, and movement must be added, although they are not achieved in the same way. System considerations include: (1) processing of sensor data to derive relevant information about the operating environment, (2) task planning based on artificial intelligence techniques to act logically on information about the environment as well as to formulate manipulator command sequences, and (3) manipulator action to carry out the task.

Of primary importance in making a robot system semiautonomous is the use of AI techniques. Much research in the field of AI has been directed toward developing an information-processing theory of intelligence. For some (computer scientists and engineers), the purpose has been to understand how machines can be made to exhibit intelligence. Others (psychologists, linguists, and philosophers) have used the computer as a tool to understand intelligent behavior in humans. A common goal has been to discover principles that all intelligent information-processors must use. This research has produced techniques in the basic areas of knowledge representation, problem solving and planning, logic, language understanding, learning and perception—areas that are especially relevant to a semiautonomous robot system.

Task planning is problem solving in a noncommutative system and, as such, requires that a good representation of the problem be selected as well as an efficient control strategy for seeking a solution. The chosen representation not only must provide techniques for modeling the environment in which the robot finds itself and for describing the process of changing one world state to another, but must also be compatible with the evolutionary development of the robot system. Because AI programming languages that have been developed are modular, changes in the knowledge base do not require extensive changes to the existing programs.

Most existing techniques and representations for generating plans were developed for purely cognitive problems and have rarely been implemented in real-time control of robots. This report summarizes work performed over the past year in which an AI-based planner was integrated with a vision system and a PUMA-560 robot to complete specified tasks involving the sorting of objects on a table. The Planner, written in PROLOG, generates the plans and monitors the execution of the plan by using feedback from the manipulator's force sensor and the vision system. Work continues on replanning and error recovery.

A general history of planners and a description of high-level languages are given to provide background for the rest of the paper.

A Historical Review of AI Planners

Planning as a problem-solving technique that produces a set of primitive operations to be carried out to achieve a goal has been the subject of much research over the past 15 years. An important early problem

solver, The General Problem Solver (GPS), was designed in 1957 by the Carnegie-RAND group: Allen Newell, C. J. Shaw, and Herbert A. Simon. Both Simon and Newell had maintained a long-term interest in human problem-solving methods. They identified a number of these problem-solving techniques using the transcripts from several experiments in which subjects were asked to "think aloud" as they were solving various kinds of puzzles. Among the techniques that were made a part of GPS was means-ends analysis, which compares the current state of the world (where we are) with the goal state (where we want to be). If these are the same, the problem is solved. Otherwise, an operator or function that will reduce the difference between the two states is selected and then applied. This sequence is continued until the current state matches the goal state.

GPS succeeded in simulating human problem solving, but only in a limited problem domain. The degree of specialized knowledge required to solve a class of problems was much greater than the designers had anticipated and made a general problem solver infeasible. However, GPS demonstrated that a machine can solve problems by functional reasoning and clarified some of the problem-solving procedures that human beings had been using all along. The techniques of GPS were to become part of a number of planners developed over the next 15 years.

At about the time GPS appeared, John McCarthy of MIT was working on a similar idea. He proposed Advice Taker, a program that would use common sense to solve a variety of problems and would interactively take advice in order to improve its performance. Unlike GPS, Advice Taker was never fully implemented. However, the idea of getting advice from an expert to help solve a problem has been used in the design of several planners and at least one speech recognition program, HEARSAY. Two important by-products of McCarthy's work on the Advice Taker were the creation of LISP and the first implementation of a time-sharing system.

In 1965 Alan Robinson described an efficient method of proving theorems in first-order predicate calculus (Robinson, 1965). AI groups at MIT, Stanford Research Institute (SRI), and the University of Edinburgh recognized that this Resolution Method of theorem proving could be used to construct problem solutions. Cordell Green developed the QA3 program to explore the use of the Resolution Method for solving state-transformation problems (Green, 1969). The world is modeled as a state space, actions as state transitions. The system attempts to prove that a state exists in which the goal condition (or theorem) is true using one set of assertions about the initial state and another set describing the effect of primitive operations on that state. The construction proof method produces the set of operations (state transitions) that would create the desired state. Applications of the program include puzzle solving, computer program generation, and robot planning.

During the mid-1960s four large robotics projects were initiated at Stanford, SRI, MIT, and the University of Edinburgh. While all the projects included visual perception and scene analysis, planning, and world modeling, their emphases were different. The robots at Stanford, MIT, and Edinburgh were manipulators, whereas SRI's Shakey was a mobile robot. The Shakey project focused on development of the problem-solving system (with limited hardware capabilities in the vision system); the other projects emphasized development of the vision system using somewhat ad hoc planners.

From these projects, it became evident that each component of the system required further research. Vision needed the development of high-resolution cameras and of methods for acquiring and understanding sensory data. The intelligent control of effectors required design of mechanical (hand-arm) devices, of optical range finders, and of special tactile, force, and torque sensors as well as development of real-world representations of the objects to be manipulated. The design and development of the planner for Shakey revealed a number of areas for further planning research including improved efficiency, handling of interacting goals, monitoring the execution of plans, error recovery and replanning, and learning. Manipulators, as well as mobile robots, required efficient algorithms for finding paths through a complex world. Since these early efforts, research has focused on developing the individual components. Only recently have the components again been integrated into robotic systems.

Green's QA3 program was the basis for a planning system within Shakey. In QA3, as in GPS, the various states of the world were completely independent; no information from one state was assumed to carry through to the next. Each operator required a large number of facts to describe completely the state of the world, some describing relationships that were changed by the action, and others (frame axioms) describing relationships that remained the same. Since most actions leave most of the world unchanged, Stanford Research Institute Problem Solver (STRIPS) was introduced to allow the system to focus its attention on the important things, the things that do change. STRIPS eliminated the frame axioms and

adopted an assumption that an action leaves all relations in the model unchanged unless specified otherwise. This assumption became known as the "STRIPS assumption." The changes were denoted using two lists for each STRIPS operator: an "add list" and a "delete list." The add list contains those relations that are always true after the action is performed, and the delete list contains those relations that are not to be true afterwards, even if they were true before. Also associated with each operator are the preconditions that must be true before the operator can be applied.

To achieve a goal, an appropriate operator is selected. Making the operator's preconditions be true then becomes the subgoals that are achieved through recursive application of the planner. This method of planning is called problem reduction and gives a hierarchical structure to plans.

In addition to finding a partial solution to the frame problem, Fikes and Nilsson sought to minimize the amount of search done in a planner based on resolution-based theorem proving by incorporating means-ends analysis to guide the selection of operators (Fikes and Nilsson, 1971). The search strategy for STRIPS is depth-first with backtracking. Although means-ends analysis restricts the number of operators that apply to a goal, there may still be several applicable operators and no way of knowing whether the subgoals of an operator can be satisfied or whether the attempt to satisfy them eventually leads to a dead end. STRIPS assumes that a goal can be completely satisfied and proceeds to fill in all the details of the plans to meet the subgoals. If one of the subgoals proves to be unsatisfiable, the work done on the plan is wasted. Therefore, this search strategy can be highly inefficient and is limited to finding plans with only a few steps.

A second problem with STRIPS is that it is a linear system and, thus, cannot solve all problems. A system based on the linear assumption expects that a goal can be achieved by first formulating plans to achieve each of the independent subgoals. The concatenation of these subplans in an arbitrary order forms a plan to achieve the goal. Because it is assumed that the subgoals do not interact, no provision is made for the interleaving of subplans.

A second version of Shakey added a plan executive component (PLANEX) to the system (Fikes et al., 1972). PLANEX monitored the execution of a plan and instigated replanning when the plan failed. Another major addition was a process for generalizing a plan produced by STRIPS. This generalized plan was stored in a tabular form called a "triangle table." This addition added flexibility to the supervision of execution since it was now possible to

(1) recognize and omit unneeded steps in the plan,
(2) reexecute a portion of the plan if necessary, and
(3) repeat an unsuccessful portion of the plan with different arguments.

Perhaps a more obvious function of the generalized plan is its use as a single macro action (MACROP). With the triangle table format, it is possible to use part or all of a MACROP as a single component in a new plan to solve a similar problem. This "learning" of plans can reduce planning time of subsequent problems and make the formulation of longer plans possible.

STRIPS, one of the first successful planners, became the basis for many planners that followed. One goal of research in the early 1970s seemed to be to overcome the limitations of STRIPS by designing planners that could solve problems with some degree of complexity or could handle interacting goals. Later planners incorporated methods for dealing both with complex problems and with interacting goals.

Attempts to improve efficiency have focused on reducing the search space by use of hierarchical planning, domain-specific information, and meta-planning. ABSTRIPS extended STRIPS by adding the capability for hierarchical planning (Sacerdoti, 1973). Although all plans have a hierarchical structure, hierarchical planners generate a hierarchy of representations of a plan in which the highest is a simplification, or abstraction, of the plan and the lowest is a detailed plan, sufficient to solve the problem. Because a complete plan is formulated at each level of the hierarchy, dead ends can be detected early in the search. A means of ignoring details that obscure or complicate a solution to a problem is also provided.

In ABSTRIPS the hierarchy is defined in terms of the criticality of the plan details. Criticality is manually assigned, and at each level only those preconditions with the current level of criticality are considered in the formation of a plan. LAWALY is a planner that has proved to be more efficient than ABSTRIPS because it combines two approaches to efficient planning (Siklossy and Dreussi, 1973). It partitions the problem-solving operators into hierarchies and constructs domain-specific procedures for each problem domain. NOAH has a certain similarity to LAWALY in that the hierarchy involves problem-solving operators.

NOAH (Nets of Action Hierarchies) abstracts problem-solving operators so that at the higher levels, the plans are made up of generalized operators. At the

lowest level, the plan consists of the primitive operators of the problem domain. NOAH uses a representation for plans called a procedural net. The net is built by adding nodes that are more specific versions of the operators represented by their parents. When the plan is completed, the procedural net is used to monitor execution.

MOLGEN, a knowledge-based program that assists molecular geneticists in planning experiments, abstracts not only the operators but also the objects in its problem space. Stefik wrote a planner for MOLGEN that extended the work of hierarchical planning to include a layered control structure for meta-planning, planning about planning (Stefik, 1980). The lowest layer is the planning space that uses the hierarchy of operators and objects. The higher levels allow MOLGEN to treat the planning process itself as another task for the planner to solve. Decisions about the design of the plan are made in the second layer, and strategies that dictate design decisions are made on the highest level.

Also concerned with meta-planning, Wilensky and Faletti included methods for changing the strategies used to formulate plans (Wilensky, 1983; Faletti, 1983). Both argued for incorporation of common sense into the planning process.

Because of the inability of linear planners to solve certain problems, a number of planners were written to investigate ways of handling such problems. HACKER was the first planner that included a strategy for doing this (Sussman, 1973). Its method was to create a plan that was "almost right" and then debug it. This debugging capability was also used to adapt an old procedure for handling problems previously solved to a new procedure to solve a new problem. WARPLAN first tried to formulate a plan using the linear assumption (Warren, 1974). If a plan cannot be found, then subgoals are selectively interleaved until a plan is found. WARPLAN is complete; that is, it will find a plan if one exists. WARPLAN was the first planner written in PROLOG.

INTERPLAN depends on the hierarchical structure of plans to solve problems it cannot solve under the linear assumption (Tate, 1975). A subgoal that conflicts with a previously achieved goal is "promoted" up a level in the hierarchical structure. Promotion and some reorderings are repeated until a plan is found. It should be noted that not all problems can be solved by INTERPLAN. Waldinger's strategy for handling interacting subgoals is to develop a plan for one subgoal and then modify that plan to achieve the second subgoal as well (Waldinger, 1981). He called his technique "goal regression."

The four programs mentioned above generated initial plans that violated ordering constraints and then fixed the plans by reordering component operators. NOAH and Stefik's MOLGEN use a "least-commitment" approach that puts off any ordering of operators until it is clear that a particular ordering is necessary to avoid conflicts. MOLGEN will not order operators until constraints are available to guide it. NOAH has "critics" that detect and correct interactions using the declarative, or plan, knowledge that is represented in the procedural net.

Until the late 1970s, planners were designed to produce complete plans. The planners of McDermott and the Hayes-Roths are based on human approaches to planning and almost never construct a complete plan (McDermott, 1978; Hayes-Roth and Hayes-Roth, 1978). McDermott sees planning and execution as interleaved processes. The planner picks a subgoal to work on according to scheduling rules. If the subgoal is a primitive, it is executed immediately. Otherwise, it is reduced to its subgoals.

The Hayes-Roths' approach is modeled after a human planning strategy of developing a plan in a piecemeal fashion; as opportunities present themselves, detailed problem-solving actions are included in the developing plan. Thus, opportunistic planning includes a bottom-up, as well as a top-down, component. The Hayes-Roths used for their planner a model developed for the HEARSAY II system for speech understanding. Knowledge sources, or experts, participate in the planning process, using a global data structure called a "blackboard" for communication. The structure of the plans is heterarchical rather than hierarchical.

Recent research has dealt with issues concerning real-time control of a robot by a planning system. Some of the issues are error recovery (Ward and McCalla, 1983); replanning (Cromarty et al., 1984); interactive planners (Wilkins, 1984); and planning using temporal logic (Allen and Koomen, 1983).

What Are High-Level Languages?

A programming language is developed to make solving a certain class of problems easier. The language provides the means for specifying the objects and operations needed to solve the problem. For example, FORTRAN was designed for numerical computing and thus provides higher-level algebraic primitives. Similarly, researchers in AI have invented

their own programming languages with features designed to handle AI problems. In fact, new ideas in AI are often accompanied by a new language in which it is natural to apply these ideas.

The kinds of problems most AI programming languages are designed to solve have arbitrary symbols as the objects to be manipulated. These symbols can stand for anything, not just numbers; by means of some data structure, relationships between symbols can be represented. IPL, one of the earliest programming languages of any kind, was the first to introduce list processing as a means of forming associations of symbols (McCorduck, 1979). IPL was created by Newell, Shaw, and Simon for their early AI work on problem-solving methods and was designed using ideas from psychology, especially the intuitive notion of association.

List processing in IPL provided not only a meaningful way of representing objects and their associations, but also a way of building data structures of unpredictable shape and size. When parsing a sentence, choosing a chess move, or planning robot actions, one cannot know ahead of time the form of the data structures that will represent the meaning of the sentence, the play of the game, or the plan of the action. Nor can the exact amount of memory that will be needed be determined ahead of time. Since the unconstrained form of data is an important characteristic of AI programs, the general goal of data representation for any AI programming language is to provide for convenient and natural representations of objects and to free the programmer from the details of memory management.

In the summer of 1956, the first major workshop on artificial intelligence was held at Dartmouth College. At this workshop John McCarthy, one of the organizers, heard a description of the IPL programming language. McCarthy realized that an algebraic list-processing language would be very useful and proceeded to implement such a language on the IBM-704 computer. This language, LISP, is the second-oldest programming language currently in widespread use (only slightly younger than FORTRAN).

In addition to its rich set of list-processing primitives, three features of LISP have contributed to its popularity among the AI community. First, LISP has a style for describing computations that is different from those of algorithmic languages such as FORTRAN or PASCAL. Instead of specifying a sequence of steps to solve a problem, LISP uses the application of functions. The function definitions are patterned after mathematical functions using lambda calculus notation. From recursive function theory, McCarthy took the idea of recursive function definitions, and LISP became the first language to support recursion.

A second important feature of LISP is that it has an interpretive execution environment that permits interactive programming. AI programs tend to have certain characteristics that greatly influence the practice of programming. First, they are big. Programmers usually try to break the system down into several discrete modules that can be written and tested separately. Often AI projects are developed incrementally, module by module. During this incremental development, not-yet-written modules may be simulated by a person interacting with the program. Also, since the development of an AI program is usually a research effort, programmers often find that the best way to develop the program is to work with it interactively—giving it a command, then seeing what happens. It was primarily this last feature that prompted McCarthy to design LISP as an interactive language.

Finally, LISP represents both functions (or programs) and data by the use of lists. Because programs and data share a common representation, it is easy to write LISP programs for handling LISP programs. For example, a LISP interpreter itself may be written in LISP. This feature also simplifies the automatic generation and modification of LISP code and the addition of extensions to the language for particular applications.

Most AI languages in use today have been designed as extensions to LISP. They offer some extra functions, data types, and control structures that augment the basic set LISP provides. Some of these are PLANNER, FUZZY, QLISP, OPS-5 and SRL. POP-2, the most common AI language in Great Britain, was developed by AI researchers at the University of Edinburgh because a good implementation of LISP was not available and because they wanted LISP-like ideas in an ALGOL-like syntax.

PROLOG, the language chosen for this project, is one popular AI language that is not an extension of LISP (Clocksin and Mellish, 1981). PROLOG (PROgramming in LOGic) is based on a first-order predicate calculus representation and is implemented as a resolution-based theorem prover. In most conventional programming languages, the programmer specifies the logic, or knowledge to be used in solving a problem, and the control, the way in which the knowledge is used. In logic programming, as advocated by Kowalski, the logic and control components of algorithms are separated, the programmer specifying only the logic part (Kowalski, 1979). A programming language that provides the means for stating what is to be done but not how it is to be done is a nonprocedural language (MacLennan, 1983). PROLOG uses a nonprocedural representation but includes procedural

semantics; the programmer provides the logic component, while PROLOG provides the control component. These procedural semantics may be manipulated to address such issues as multiple answers, control of backtracking, and efficiency.

A program in PROLOG is structured like the statement of a mathematical theorem and is divided into three parts. First is a number of general principles (or inference rules) that define the problem domain. The second part is a statement of a number of particular facts. This part defines the relations among the objects and is often referred to as the data base. The third part is the statement of the goal (or the problem to be solved) as a theorem to be proved. Proving the theorem generates the answer.

In a world consisting of colored blocks to be manipulated, for example, we would have facts about the blocks and their relationships, such as

```
is_on(blue, red).
clear_top(blue).
is_on(red, table).
```

(Note: Variables in PROLOG begin with uppercase letters, and constants begin with lower case.)

The inference rule that says if BlockA is not the same block as BlockB, and BlockA can be moved to the top of BlockB, then is_on(BlockA, BlockB) is true and is stated

```
is_on(BlockA, BlockB) :-
        not(BlockA = BlockB),
        move(BlockA, BlockB).
```

To prove the goal

```
:- is_on(Block, red),
```

the fact is_on(blue, red) could be used: There exists a block that is on the red block—the blue block. Or the inference rule could be used for the proof. This would find a block that is not the red one and, in proving the subgoal, move(BlockA, BlockB), cause that block to be moved to the top of the red block as a side-effect of the proof procedure. The proof of is_on(Block, red) is completed, and the result is a sequence of moves, or proof statements, that place the block in the proper position.

Thus, programs are expressed in the form of propositions that assert the existence of the desired result. The theorem prover must construct the desired result to prove its existence. In a nonprocedural representation like PROLOG, the program states what result is wanted without specifying how to get it. Because the program sets forth the relations rather than the flow of control, the programmer is relieved of the responsibility for working out the steps of an algorithm and specifying their order. This also means that showing a program is correct is greatly simplified because only the logic component of a program must be dealt with.

In examining PROLOG for the features that were stated as important to AI programming, we find that PROLOG has most of the features. PROLOG provides for symbol manipulation and for the defining of data structures to handle the unpredictable shape and size of the data. It can be executed interactively, and the program and data share a common representation.

The program and data are both represented by clauses of the general form

```
<head> :- <body> .
```

If the <head> is omitted, it is considered a goal; if the :- <body> is omitted, it is considered a fact. Both the <head> and the <body> are composed of predicates of the form

```
predicate(term1, term2,...,termn),
```

with a term representing individual objects in the problem domain and the predicate defining a relation among the terms. In PROLOG a clause must be in Horn clause form; <head> has at most one predicate but <body> may have any number.

PROLOG does not have a fixed set of data structure constructors. Rather, a data structure is defined implicitly by giving a description of the properties of its operations. Thus, all data types are inherently abstract data types. This is the nonprocedural approach to data structures.

In addition to the features already mentioned as being important, there are other features that make PROLOG a convenient language to use. First, there is no distinction between input and output variables so that a single predicate may function in several different ways. Consider the predicates from the previous example and the goal

```
:- is_on(X, Y).
```

If neither X nor Y were set to a value, then the goal would be proved by finding

```
X = blue, Y = red
X = red, Y = table.
```

If X is set to the value blue, this goal would use the fact is_on(blue, red) and find

Y = red.

With Y set to table, it would find

X = red.

One way of looking at this situation is that in PROLOG, a program can be run either forward or backward, as needed.

Finally, a program written in PROLOG is very readable. Since programs are described in terms of predicates and objects of the problem domain, programs are almost self-documenting. This characteristic promotes clear, rapid, accurate programming.

Task Planning in a Real-World Environment

These planning concepts have been applied to an actual manipulator system termed the Sensor-Driven Robot Systems Testbed, located at Sandia National Laboratories (Donohoe, 1985). The Sensor-Driven Robot Systems Testbed, as configured for this investigation, includes a PUMA-560 robot manipulator with an Astek Corporation force-sensing wrist capable of resolving three independent axial forces and the associated torques about those axes. In addition, a vision-sensor subsystem (currently limited to binary vision) provides object location and orientation information.

The "world" of the Sensor-Driven Robot Systems Testbed currently consists of a table top supporting discrete objects for manipulation. In this study, 50-mm cubes were manipulated. Each cube was uniquely labeled with spots (as dice) to allow block identification. A photograph of the Sensor-Driven Robot Systems Testbed is shown in Figure 1.

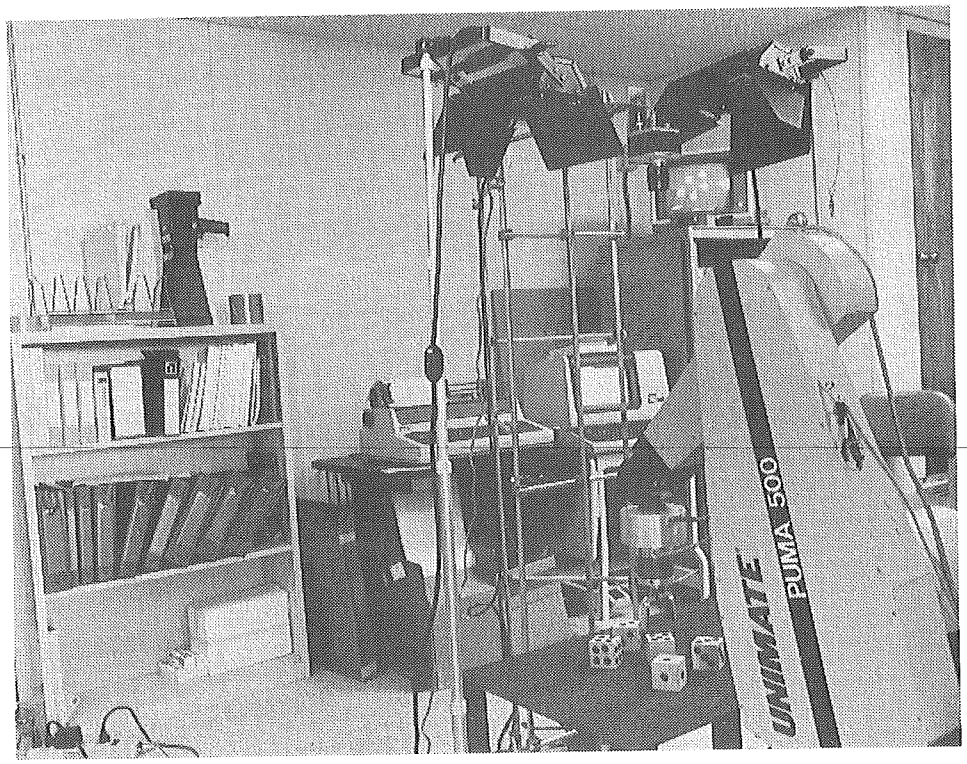


Figure 1. Sensor-Driven Robot Systems Testbed

A CCD camera is shown mounted above the table (painted flat black to provide high contrast), with flood lamps for scene illumination. The large cylindrical object mounted on the wrist of the PUMA manipulator is the Astek Corporation force-sensing wrist. In the background is the computer system dedicated to image processing. A detailed description of the image-processing subsystem is presented by Daily and Donohoe (1985). The robot controller and supervisory computers are not shown in Figure 1.

The system controlling the Sensor-Driven Robot Systems Testbed consists of a hierarchical network of computers, as illustrated in Figure 2. In a typical situation, the vision subsystem communicates the initial state of the world (i.e., what objects are on the table top and their respective locations and orientations). This information is communicated to the Local Supervisor, which coordinates interactions with a VAX-11/780 computer resident at the University of New Mexico. The Local Supervisor is the primary point of interaction for the operator. The Local Supervisor communicates the initial world state and the desired world state (provided by the operator) to the PROLOG-based Planner resident on the VAX-11/780. The Planner then formulates a series of robot movements (in the VAL-II language of the PUMA-560) to achieve the goal state configuration. These VAL-II commands are communicated back to the Local Supervisor, which in turn communicates with the Robot Controller.

Simple sensory-based path control and error detection is provided by the force-sensing wrist. For example, the force-sensing wrist is used to determine contact in the vertical direction during placement of an object either on the table surface or upon another object. To prevent an object's being dropped, it is released only after a prescribed force ($\sim$1 N) in the vertical direction is exceeded. Error situations can be detected by the appearance of unexpected forces at the force-sensing wrist. One error situation that has been effectively handled has been small errors in object locations. If the inherent inaccuracies of the vision system provide erroneous locations for the objects to be manipulated, the robot gripper will encounter an object at an unexpected height above the table top. Typically, only one jaw of the two-jaw gripper will strike the object, thus generating a torque at the robot's wrist. Since the torque is a vector quantity, the Local Supervisor can determine the proper direction to move the gripper in order for the gripper to span the object. The Local Supervisor, using this information, generates a correction trajectory for the gripper, and a new attempt is made to grasp the object. The process can be repeated any number of times to compensate for object location misinformation.

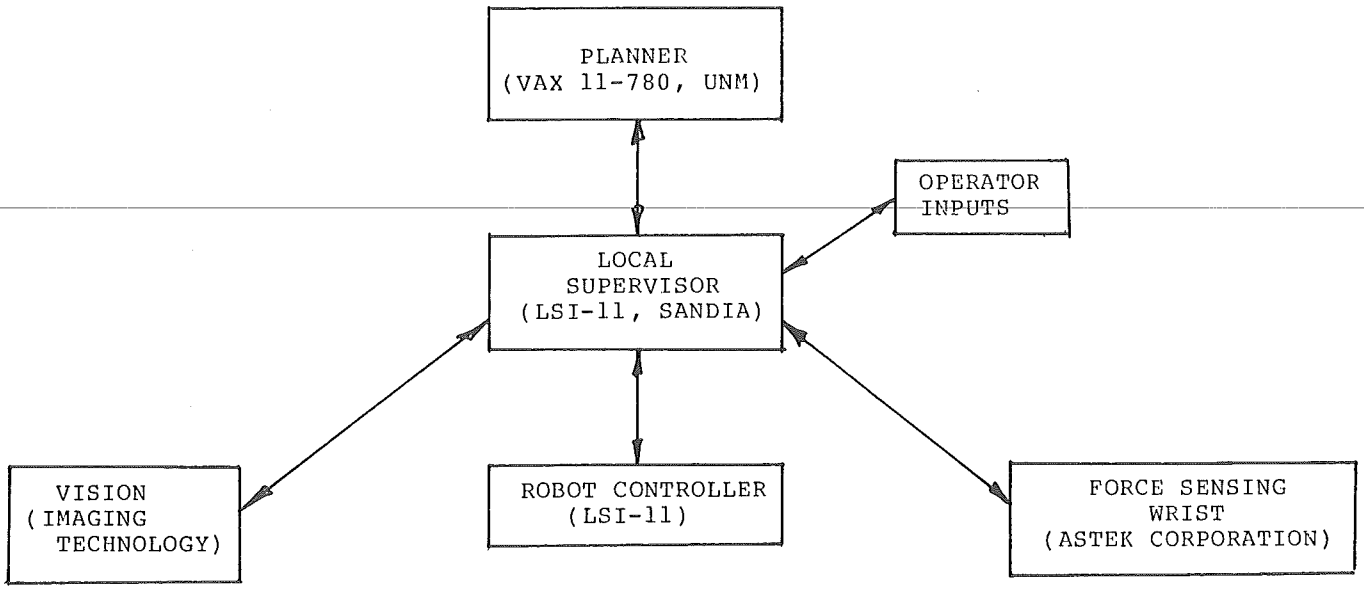
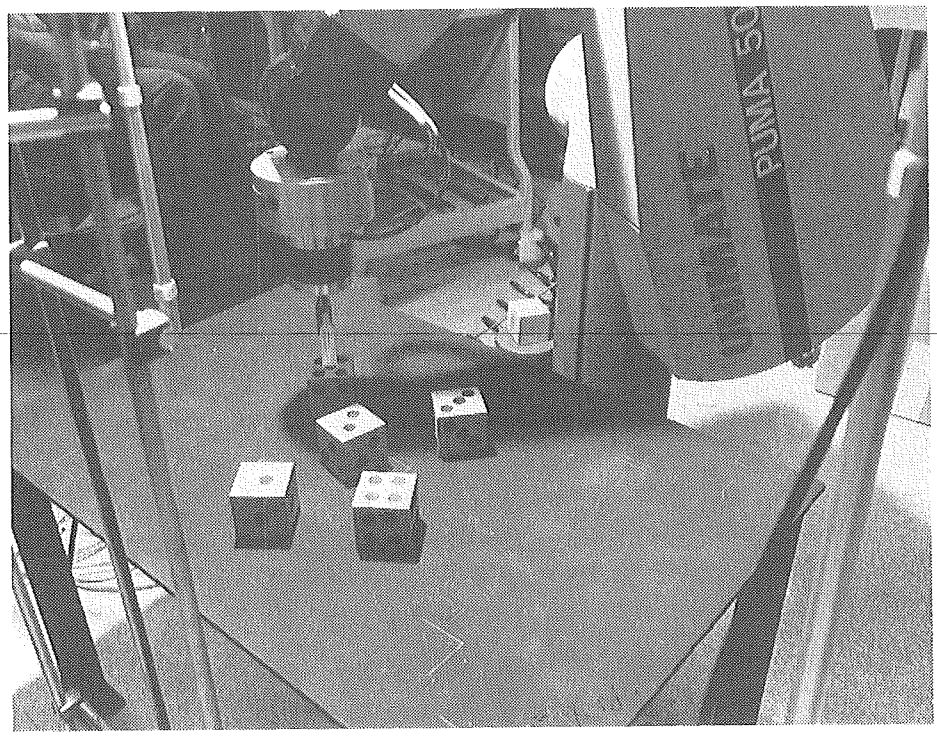


Figure 2. Computer Hierarchy for Control of Sensor-Driven Robot Systems Testbed

Object manipulation by the PUMA-560 in the Sensor-Driven Robot Systems Testbed is controlled by the Planner. The Planner is based on ideas from STRIPS and WARPLAN. In these codes, plans consist of a sequence of actions, and each action is composed of a triple: a list of preconditions necessary for performing an action, the action itself, and the post-conditions that describe the changes in the state of the world once the action is performed. Unlike STRIPS, a complete plan is never formulated in our planner, but the planning and the execution of the plan are interleaved. If a task is a primitive action, it is executed immediately. Otherwise, the task is reduced to achieving subtasks. This allows the Planner to react more efficiently to unexpected consequences of actions.

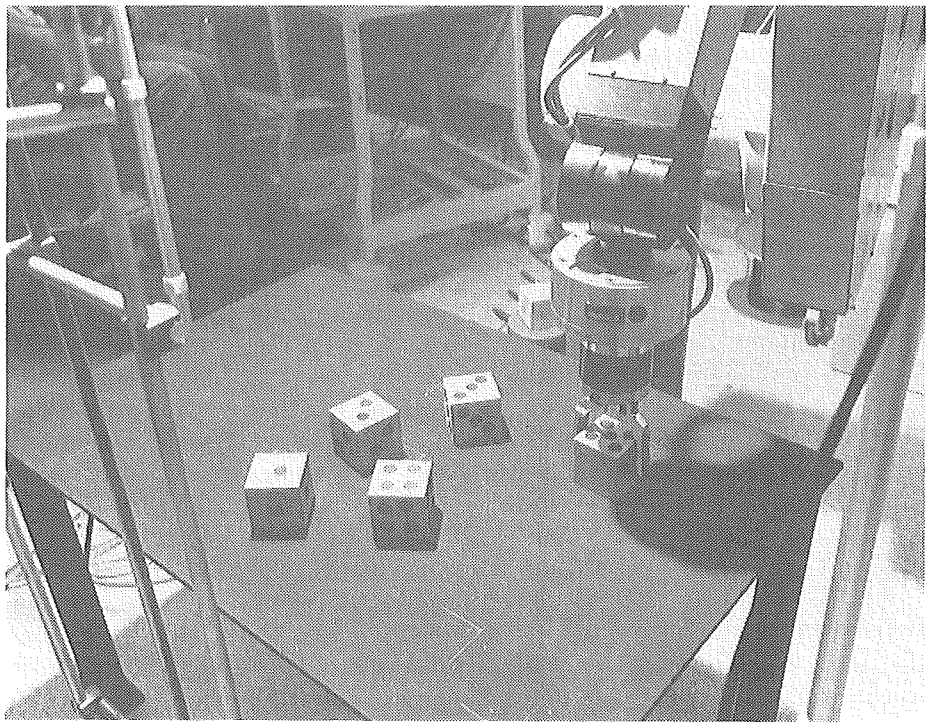
After establishing the initial positions of the blocks with the binary vision system, the Planner constructs its world model. The world model is a set of predicates that describe the state of the world and include the position of the arm, the number of blocks, where they are on the table, which ones are parts of stacks, and which ones have clear tops. Once the state of the world is determined, the Planner enters into a dialogue with the operator to determine the task specification and then starts formulating a plan for the sequence of moves the robot must make to complete the task. Upon completion of the task, the goal state becomes the initial world state for future block manipulations. Selected PUMA-560 movements generated by the Planner to create a stack of five blocks at an operator-requested location are shown in Figure 3.

Commands available are to STACK blocks, to UNSTACK blocks, and to SEE the state of the world, that is, to list the PROLOG predicates that comprise the world model on the terminal. The operator may choose to create a stack of blocks at a specific location on the table or on top of a specific block. The Planner reasons whether the requested stacking is possible, considering the physical constraints imposed by the size of the manipulator's gripper. If the specified location is too close to an existing stack, the Planner will ask for further instructions; a new location may be specified or the offending stack may be moved. In generating the commands for moving blocks, the Planner performs simple obstacle avoidance by instructing the arm to lift a block over a stack if necessary to avoid collision with the stack. In unstacking, the Planner finds the closest possible location to place the block on the table, keeping in mind the physical constraints.



a

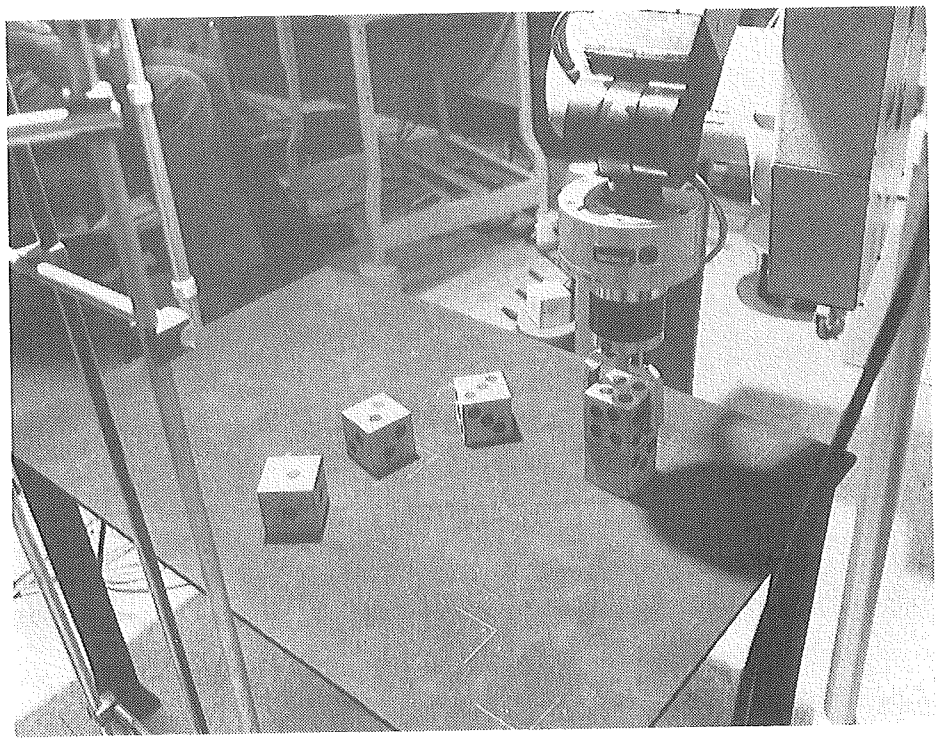
Figure 3. PUMA-560 Manipulation Sequence to Accomplish Block Stacking



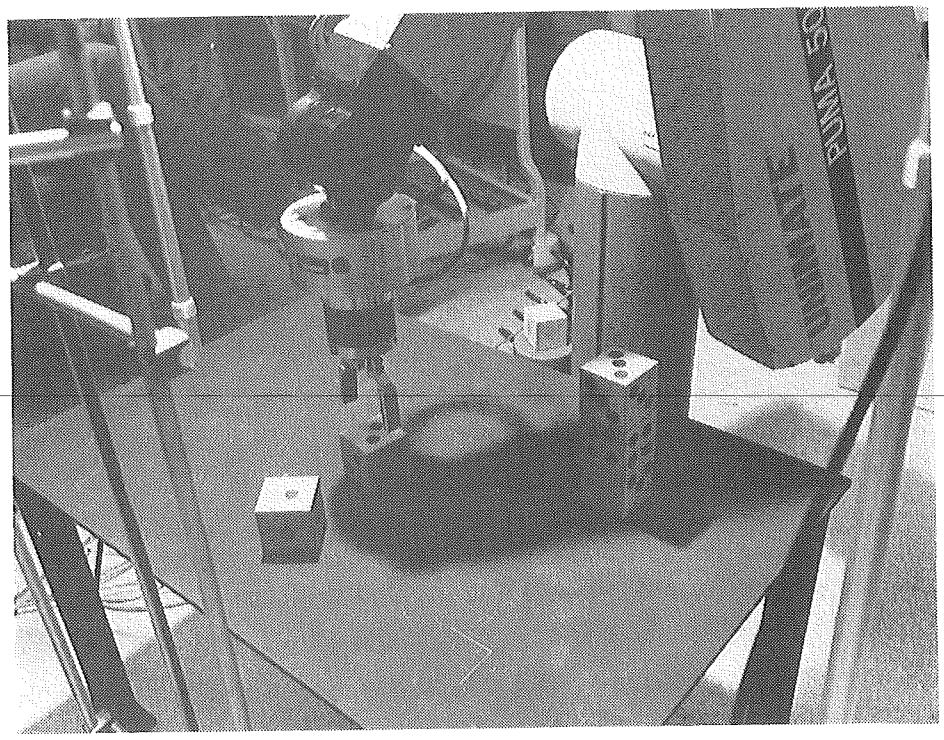
b

c

Figure 3. Continued

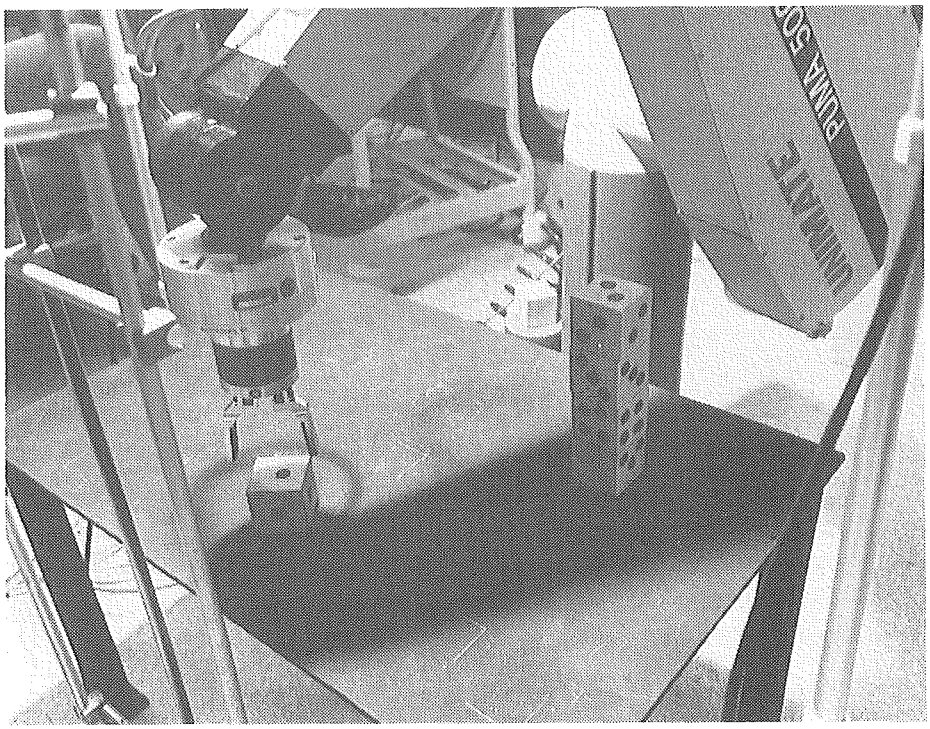


d



e

Figure 3. Continued



f

g

Figure 3. Concluded

Future Work

As it now exists, the Sensor-Driven Robot Systems Testbed represents a starting point for Planner development. In particular, intelligent error recovery is an important area for future work. An example of intelligent error recovery would involve locating and picking up an inadvertently dropped object. If the robot manipulator dropped an object and, as a result, could not complete a task, the Planner should not only, for example, request an update on the state of the world from the vision system but also indicate what part of the visual scene must be analyzed and which object is to be located. This can be very important in relatively complex scenes that require extensive analysis.

An additional intelligent error recovery capability of interest is active interaction between the robot manipulator and the primary sensors, such as vision. If, for example, the vision subsystem cannot resolve a scene completely because of touching or hidden objects (e.g., one is behind another), the Planner, with its knowledge of the history of the world and the objects being manipulated, should be able to direct the robot manipulator to move the camera to a location helpful in resolving uncertainties. In addition, rather than requesting that the vision system perform a complete scene analysis, the Planner, with its knowledge of what features it is looking for, could direct a selected search of the visual image to enhance system response time.

Thus, the general thrust for future work will involve sensory enrichment of the system with more intimate interactions between the sensory subsystems and manipulator, with guidance from the Planner to resolve incomplete or inconsistent knowledge.

References

Allen, James F., and Koomen, Johannes A., "Planning Using a Temporal World Model," *Proc International Joint Conf on Artificial Intelligence (IJCAI)*, 1983, pp 741-47.

Clocksin, W. F., and Mellish, C. S., *Programming in Prolog*, Springer-Verlag, 1981.

Cromarty, A. S.; Shapiro, D. G.; and Fehling, M. R.,"'Still planners run deep': Shallow reasoning for fast replanning," *SPIE Applications of Artificial Intelligence*, 1984.

Daily, M. J., and Donohoe, G. W., *Visual Control of a Robot Arm for Object Manipulation*, SAND85-0681 (Albuquerque, NM: Sandia National Laboratories, June 1985).

Donohoe, G. W., "A Testbed for Sensory Control of Intelligent Machines," 6th IASTED International Symp on Robotics and Automation, May 1985.

Faletti, Joseph, "PANDORA—A Program for Doing Commonsense Planning in Complex Situations," *AAAI*, 1983, pp 185-88.

Fikes, Richard E. and Nilsson, Nils J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence 2*, 1971, pp 189-208.

Fikes, Richard E.; Hart, Peter E.; and Nilsson, Nils J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence 3*, 1972, pp 251-88.

Green, Cordell, "Application of Theorem Proving to Problem Solving," *IJCAI*, 1969, pp 219-39.

Hayes-Roth, Barbara, and Hayes-Roth, Frederick, *Cognitive Processes in Planning*, Rand Corp Report R-2366-ONR, 1978.

Kowalski, Robert, "Algorithm = Logic + Control," *Comm of ACM* 22:7, 1979.

MacLennan, Bruce J., *Principles of Programming Languages: Design, Evaluation and Implementation*, Holt, Rinehart & Winston, 1983.

McCorduck, P., *Machines Who Think*, W.H. Freeman & Co, 1979.

McDermott, Drew, "Planning and Acting," *Cognitive Science 2*, 1978, pp 71-109.

Robinson, J. A., "A Machine-oriented Logic Based on the Resolution Principle," *J ACM* 12:1, 1965.

Sacerdoti, Earl D., "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence 5*, 1973, pp 115-35.

Siklossy, L., and Dreussi, J., "An Efficient Robot Planner Which Generates Its Own Procedures," *IJCAI*, 1973, pp 423-30.

Stefik, Mark J., "Planning with Constraints," Stanford University Ph.D. Thesis, 1980.

Sussman, Gerald J., *A Computational Model of Skill Acquisition*, MIT AI Lab Technical Report AI-TR-297, August 1973.

Tate, Austin, "Interacting Goals and Their Use," *IJCAI*, 1975, pp 215-18.

Waldinger, Richard, "Achieving Several Goals Simultaneously," *Readings in Artificial Intelligence*, B. Webber and N. Nilsson, ed, Tioga Pub Co, 1981.

Ward, Blake, and McCalla, Gordon, "Error Detection and Recovery in a Dynamic Planning Environment," *AAAI*, 1983, pp 172-75.

Warren, David H. D., "WARPLAN: A System for Generating Plans," Dept Computational Logic, Memo No. 76, Univ of Edinburgh, 1974.

Wilensky, Robert, *Planning and Understanding: A Computational Approach to Human Reasoning*, Addison-Wesley, 1983.

Wilkins, D. E., "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence* 22:3, 1984, pp 269-301.