R. Charleene Lennox, George F. Luger, & Raymond W. Harrigan *Artificial Intelligence Based Control of a Sensor-Driven Machine.* Proceedings of Conference on Computers and Communications, Sponsored by IEEE Computer Society, 1985

Artificial Intelligence Based Control of a Sensor-Driven Machine

by

R. Charleene Lennox
George F. Luger
Department Of Computer Science
University of New Mexico
and
Raymond W. Harrigan
Intelligent Machine Systems Division
Sandia National Laboratories
Albuquerque, New Mexico

USEnet: ...ucbvax!unmvax!robot
ARPAnet: unmvax!robot@berkeley

Abstract

This paper is divided into four sections. The first is an introduction to the research area. The second is a brief overview of the history of "planning" in Artificial Intelligence. 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 in solving "blocks" problems.

1. Introduction

Intelligent machine systems will necessarily require the integration of sensors and artificial intelligence 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 thus 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 so must the capabilities of sight, touch, reasoning and movement be added, although not necessarily achieved in the same way. System considerations include processing of sensor data to derive relevant information about the operating environment, task planning based on artificial intelligence techniques to logically act on information about the environment as well as to formulate manipulator command sequences, and manipulator action to carry out the task.

Of primary importance in making a robot system semiautonomous is the use of artificial intelligence (AI) techniques. A great deal of research in the field of AI has been directed toward the development of 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 in understanding 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 which are especially relevant to a semiautonomous robot system.

Task planning is problem solving in a noncommutative system and, as such, requires selecting a good representation of the problem as well as an efficient control strategy for searching for 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. AI programming languages have been developed that are modular so that changes in the knowledge base do not require extensive changes to the existing programs.

While techniques and representations for generating plans exist, most 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 artificial intelligence 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 using feedback from the manipulator's force sensor and the vision system. Work is continuing 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.

2. An Historical Review of AI Planners

Planning, as a problem solving technique which produces a set of primitive operations to be carried out to achieve a goal, has been the subject of much research over the last 25 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 had a long-time 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 which were made a part of GPS is means-ends analysis which compares the current state of the world (where we are) with the goal state (where we want to be). If they are

the same, the problem is solved. Otherwise, an operator, or function, which 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 amount of specialized knowledge required to solve a class of problems 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 a part of a number of planners developed over the next 25 years.

At about the time GPS appeared John McCarthy, of MIT, was working on a similar idea. He proposed Advice Taker, a program which would use common sense to solve a variety of problems and would interactively take advice in order to improve its performance. This idea of getting advice from an expert to help solve a problem has been used in the design of several planners. 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 published a paper [13] describing an efficient way of proving theorems in first-order predicate calculus. AI groups at MIT, Stanford Research Institute (SRI) and the University of Edinburgh recognized that the Resolution Method of theorem proving could be used to construct problem solutions. Cordell Green [7] developed the QA3 program to explore the use of the Resolution Method for solving state-transformation problems. The world is modelled as a state space and actions as state transitions. The system attempts to prove that there exists a state in which the goal condition (or theorem) is true using a set of assertions about the initial state and another set that describes the effect of primitive operations on the state. The construction proof method produces the set of operations (state transitions) that would create the desired state. Applications of the program included puzzle solutions, computer program generation and robot planning.

During the mid-60's four large robotics projects got underway at Stanford, SRI, MIT and the University of Edinburgh. While all the projects included visual perception and scene analysis, planning, and world modelling, the emphasis was different. The robots at Stanford, MIT and Edinburgh were manipulators while SRI's Shakey was a mobile robot. The Shakey project focused on developing the problem-solving system with limited hardware capabilities in the vision system while the others emphasized development of the vision system using somewhat ad hoc planners. From these projects it became evident that each component of the system required forther research. Vision needed the development of high resolution cameras and of methods for acquisition and understanding of 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. During the next few years research focused on developing the individual components, and it has been only recently that the components have 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 completely described the state of the world, some describing relationships which were changed by the action, and others (frame axioms) describing relationships which remained the same. Since most actions leave most of the world unchanged, STRIPS (Stanford Research Institute Problem Solver) was introduced to the 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 know 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 which are always true after the action is performed and the delete list contains those relations that may not be true afterwards even if they were true before. Also associated with each operator are the preconditions which 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 which 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 [5] 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. The search strategy for STRIPS is depth-first with backtracking. Although this 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 form a plan to achieve the goal. It is assumed that the subgoals do not interact, so no provision is made for the interleaving of subplans.

A second version of Shakey [6] added a plan executive component (PLANEX) to the system. 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 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
(3) repeat an unsuccessful portion of the plan with different arguments.

Perhaps, a more obvious function of the generalized plan is 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, as one of the first successful planners, became the basis for many planners which followed. One goal of research in the early 70's seemed to be to overcome the limitations of STRIPS by designing planners which could solve problems with some degree of complexity or could handle interacting goals. Later planners incorporated methods for dealing with both problems.

Attempts to improve efficiency have focused on reducing the search space by use of hierarchical planning, domain-specific information and meta-planning. ABSTRIPS [14] extended STRIPS by adding the capability for hierarchical planning. 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 details of the plan with only those preconditions with the current level of criticality considered at each level in the formation of a plan. LAWALY [15] proved to be more efficient than ABSTRIPS because it combined two approaches to efficient planning. It partitioned the problem-solving operators into hierarchies, and it constructed domain-specific procedures for each problem domain. NOAH has a certain similarity to LAWALY in that the hierarchy involved problem- solving operators.

NOAH (Nets of Action Hierarchies) abstracted problem-solving operators so that at the higher levels the plans are made up of generalized operators. At the lowest level the plan is made up 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 which are more specific versions of the operators represented by their parents. When the plan is completed, the procedural net is used to monitor execution.

Stefik [16] wrote a planner for MOLGEN which abstracted both operators and the objects in its problem space and which extended the work of hierarchical planning to include a layered control structure for meta-planning, planning about planning. The lowest layer is the planning space which 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 using strategies that dictate decisions about the design of the plan.

Also concerned with meta-planning Wilensky [21] and Faletti [4] included methods for changing the strategies used to formulate plans. Both argued for incorporation of commonsense 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 problems with for doing this. Sussman in HACKER, Warren (WARPLAN [20]), Tate (INTERPLAN [17]), and Waldinger [18] used similar strategies which first tried to formulate a plan using the linear assumption. If the generated initial plan violates ordering constraints, the plan is fixed by debugging the "almost right" plan or by reordering component operators. WARPLAN was the first planner written in PROLOG. It is also complete; that is, it will find a plan if one exists. NOAH and Stefik's MOLGEN use a "least-commitment" approach which 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" which detect and correct interactions using the declarative, or plan, knowledge the is represented in the procedural net.

Until the late 70's planners were designed to produce complete plans. The planners of McDermott [12] and the Hayes-Roths [8] are based on human approaches to planning and almost never construct a complete plan. 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 modelled 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 a model developed for the HEARSAY II system for speech understanding for their planner. Knowledge sources, or experts, participate in the planning process, using a global data structure call a "blackboard" for communication. The structure of the plans are 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 [19], replanning [3], interactive planners [22], and planning using temporal logic [1].

3. What Are High-Level Languages?

A programming language is developed to make the solving of a certain class of problems easier. To accomplish this 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. 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 that 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, and, by means of some data structure, relationships between the 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. IPL [11] 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, respectively. Nor can it be determined ahead of time the exact amount of memory that will be needed. Since the unconstrained form of data is an important characteristic of AI programs, the general goals 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. At this workshop John McCarthy, one of the organizers of the workshop, 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 in present widespread use (only slightly younger than FORTRAN).

There are three features of LISP, besides the rich set of list-processing primitives, that 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 become the first language to support recursion.

A second important feature of LISP is that it has an interpretive execution environment which 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 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, the LISP interpreter is itself 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 [2], the language chosen for this project, is one popular AI language which is not an extension of LISP. 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 [9], the logic and control components of algorithms are separated with the programmer only specifying the logic part. A programming language which provides the means for stating what is be done, but not how it is to be done, is a nonprocedural language [10]. PROLOG uses a nonprocedural representation but includes a procedural semantics; the programmer provides the logic component while PROLOG provides the control component. This 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, we have a number of general principles (or inference rules) that define the problem domain. The second part is 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 our blocks world program we 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 upper case letters and constants begin with lower case.)
There is an inference rule that says that 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:

```
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 which is on the red block -- the blue block. Or the inference rule could be used for the proof. This would find a block which 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 a 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 nonprocedureal representation like PROLOG the program states what result is wanted without specifying how to get it. The program sets forth the relations rather than the flow of control, and so the programmer is relieved of the responsibility for working out the steps of an algorithm and specifying their order. This also means that showing that 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 doesn't 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 nice 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 is that in PROLOG a program can be run either forward or backwards as needed.

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

4. Task Planning for Robot Manipulation

Current robot systems are designed to perform a preprogrammed sequence of operations. The programming of the sequence may be done using a teach pendant or coding the operations in the robot's control language. Either method can be long and tedious and produces a program which is suitable for one task and which cannot handle unexpected occurrences. The integration of an artificial intelligence based planner and appropriate sensors in the system yields an "intelligent" robot. This robot can perform those tasks which are drawn from its set of primitive actions and are in a specified environment, as well as recover from errors which may occur. The investigation of the intelligent, sensor-driven control of a robot manipulator is the focus of a research project conducted jointly by Sandia National Laboratories and the University of New Mexico departments of Computer Science and Electrical and Mechanical Engineering. The long-range goal is to develop techniques for performing human-like maintenance tasks in hostile environments such as within a nuclear reactor. The project provides a test bed for ideas in an environment which is simpler and more traditional but still allows for interesting research in the areas of sensing, planning, error recovery, and controlling a robot.

The domain chosen for the investigation is the classic "Blocks World" of Terry Winograd. Blocks World consists of five 50 mm cubes on a table. Each block is identified by the number of black spots on it (like playing dice). The Planner, using sensory feedback from the 2-dimension vision system and the force sensor on the robot arm, generates a sequence of commands to accomplish the stacking or unstacking of the blocks as specified by the operator, and communicates the commands to the robot.

The components of the system are the Vision System, the Controller for the robot manipulator, the Supervisor and the Planner . A solid state black and white television camera is suspended over the table, looking down upon the blocks. The signal from this camera goes to an LSI-11/73 microcomputer which is the Vision processor. Next to the table is a PUMA 560 robot with a Unimate Controller and a force-sensing wrist. VAL-II is the command language of the Unimate Controller. The Supervisor acts as a communications controller and resides on an LSI-11/73. The Planner program which is written in PROLOG resides on the Research VAX at UNM and communicates with the Supervisor via a telephone link.

When the system starts up, the Vision System analyzes the scene, and locates and identifies the blocks. It then reports the locations of the vertices to the Planner by way of the Supervisor. The Planner uses this information to construct its World Model. The World Model is a set of predicates which describes the state of the world and includes the position of the arm, the number of blocks, where they are on the table, which ones are parts of stacks and which ones are 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 formulates a plan for the sequence of moves the robot must make to complete the task. It then generates commands to the Supervisor which translates them into VAL-II commands and sends them to the Controller. The force sensor is used to verify the locations of the blocks and to adjust to slight errors in positioning.

The Planner is based on traditional AI approaches to planning especially using ideas from STRIPS and WARPLAN. In these programs plans are made up 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 postconditions which describe the changes in the state of the world once the action is performed. Unlike STRIPS a complete plan is never formulated, but 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.

Commands available in Blocks World 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 STACK blocks on 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.

The Planner currently will handle either a 2-D or 3-D representation of the blocks world, the difference being that a 2-D system uses the x, y coordinates of four vertices to define a block while the 3-D representation has the x, y, and z coordinates of all eight vertices. Using the 3-D representation it is possible for the Planner to reason about the blocks that make up a stack and the location of blocks that can't be seen by a 2-D vision system.

In real-world domains things do not always proceed as planned, e.g., the robot drops a block or the air hose on arm knocks over a stack. Therefore, it is desirable to provide execution-monitoring techniques and the capability of replanning. At present only the simplest kind of error recovery has been implemented. The Controller communicates to the Planner that it was not able to move a block as directed by

the Planner. The reason for this may be that manipulator sensed that it had dropped the block while moving it or that the manipulator did not find the block where the Planner thought it was located. The Planner then asks the Vision System for a new scene analysis, updates the World Model and formulates a new plan to complete the goal task given the current state of the world.

Future plans for research call for increasing the capabilities in several areas. First, the various components need to be integrated into one computer system. At present the Planner's only way of communicating to the Supervisor the sequence of commands is by writing them to a file. Once the Planner and Supervisor are on the same computer they should be able to communicate in a more straightforward manner.

In the area of sensors, we plan to add three-dimensional vision and to include different kinds of objects. This will require a more complicated world model and will allow the building of structures other than stacks. Because of the various shapes of the objects the Planner must then plan a strategy for grasping the object. The addition of tactile sensors for identifying objects by touch and hand-like grippers for more dextrous gripping is also planned.

More sophisticated error recovery is an important area of expansion for the Planner. The Planner will need to be able to know, not only when a block has been dropped, but also when a stack has been knocked over or fallen over. This will require consulting the sensors, both vision and force, more frequently. The Vision System and Planner will have to deal with incongruous information such as touching or overlapping blocks which appear as one block and with incomplete information such as the inability of the camera to see some objects. A further goal for intelligent recovery from errors is to guide the focus of the camera with the Planner. This would let the Planner conjecture where the source of a problem might lie and turn (or even relocate) the camera to study that part of the world.

In addition to improving error recovery, there is also the consideration of whether it is possible to use part of the previous plan. This becomes increasingly important in complex domains where the amount of time it takes to formulate a plan becomes an important factor.

Acknowledgments

We would like to thank Greg Starr, Alan Christianson and the other members of the Intelligent Machine Systems Division. We would especially like to thank Sandia National laboratories for their support of this research.

Bibliography

(1) Allen, J.F., and Koomen, J.A., Planning Using a Temporal World Model, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 1983.

(2) Clocksin, W.F., and Mellish, C.S., *Programming in Prolog*, Springer-Verlog, 1981.

(3) 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.

(4) Faletti, Joseph, PANDORA - A Program for Doing Commonsense Planning in Complex Situations, *AAAI*, 1983.

(5) Fikes, R.E., and Nilsson, N.J., STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence 2*, 1971.

(6) Fikes, R.E., Hart, P.E., and Nilsson, N.J., Learning and Executing Generalized Robot Plans, *Artificial Intelligence 3*, 1972.

(7) Green, C., Application of Theorem Proving to Problem Solving, *IJCAI*, 1969.

(8) Hayes-Roth, B. and Hayes-Roth, F., *Cognitive Processes in Planning*, Rand Corp. Report R-2366-ONR, 1978.

(9) Kowalski, R., Algorithm = Logic + Control, *Comm. of ACM*, 22:7, 1979.

(10) MacLennan, B.J., *Principles of Programming Languages: Design, Evaluation and Implementation*, Holt, Rinehart & Winston, 1983.

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

(12) McDermott, D., Planning and Acting, *Cognitive Science*, 2, 1978.

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

(14) Sacerdoti, E.D., Planning in a Hierarchy of Abstraction Spaces, *Artificial Intelligence 5*, 1973.

(15) Siklossy, L., and Dreussi, J., An Efficient Robot Planner Which Generates Its Own Procedures, *IJCAI*, 1973 . .

(16) Stefik, M.J., *Planning with Constraints*, Stanford University Ph.D. Thesis, 1980.

(17) Tate, A., Interacting Goals and Their Use, *IJCAI*, 1975.

(18) Waldinger, R., Achieving Several Goals Simultaneously, *Readings in Artificial Intelligence*, B. Webber and N. Nilsson, eds., Tioga Pub. Co., 1981.

(19) Ward, B., and McCalla, G., Error Detection and Recovery in a Dynamic Planning Environment, *AAAI*, 1983.

(20) Warren, D.H.D., *WARPLAN: A System for Generating Plans*, Dept. Computational Logic, Memo No. 76, U. of Edinburgh, 1974.

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

(22) Wilkins, D.E., Domain-independent Planning: Representation and Plan Generation, *Artificial Intelligence*, 22:3, 1984.