THE UNIVERSITY OF ALBERTA

A First Order Logic Robot Planning System

by

(C) James G. Borynec

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Computing Science

EDMONTON, ALBERTA

Fall, 1987

ISBN 0-315-40904-5

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR James G. Borynec

TITLE OF THESIS A First Order Logic Robot Planning System

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall, 1987

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) [illegible]

PERMANENT ADDRESS:

10433 - 84 St

Edmonton, ALTA

T6A 3R3

DATED 20 August 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled A First Order Logic Robot Planning System submitted by James G. Borynec in partial fulfilment of the requirements for the degree of Master of Science.

[illegible]
Supervisor

[illegible]

Jia-Huai You

Roger W Toogood

Date 20 Aug 87

Abstract

A planning system is a major component of any independent robot system. This thesis describes a first order predicate calculus framework which allows the statement and synthesis of robot plans. This framework is a reduction of Rosenschein's (1981) and Kautz's (1982) work in dynamic logic, but has the potential to be more flexible. It is an attempt to provide some middle ground between the theoretical approach of dynamic logic and the more pragmatic STRIPS-like planners. An implementation of a planner is presented within this framework, which relies to some extent on default rules for carrying information forward from one state to the next, in addition to carrying out deduction. The planner synthesizes plans suitable for the University of Alberta's Kato-Heron robot.

Proem

I know what you're thinking about," said Tweedledum; "but it isn't so, nohow."

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll

Through the Looking-Glass

Table of Contents

1. Introduction

Robotic research and Artificial Intelligence are intertwined. Raphael (1970) states that it is perfectly natural to involve AI with robotic research. Furthermore, interest in the development of robot like devices is both a natural consequence of past developments and a necessary stimulus to future research.

And yet, after some initial success, research efforts have diverged into two separate fields of study, one field dealing with real-world oriented robotic research, emphasizing low level control schemes, and the other field specializing in generalized intelligence research. Inoue (1985) states that there is a significant gap between AI and robotics in spite of the necessity of communication and integration between these fields. A major motivation for the U of A robot project is to form a tiny part of the bridge between these two fields.

One important advantage of robotics over virtually any other AI research domain is that the robot must deal with the physical world, thereby placing the roles of knowledge representation, programming languages, computer configurations, etc. into clear perspective. Another feature of robotics is that, because actions performed in the real world can never be predicted with absolute certainty, any flexible robot system must have both a general purpose planning system and some sensory pattern recognition capability. The planning system will allow the robot to function in the world, while the sensory capability will inform the robot when its plans have gone awry, as well as to serve to establish facts about the world.

This thesis presents a first order function free logic in clause form which allows the statement of robot plans. This logic is a reduction of Rosenschein's (1981) and Kautz's (1982) work in dynamic logic. Propositions and actions have time parameters and are ordered by a simple syntactic notion of time ordering. The general form of the planning problem is to find a sequence of actions which, together with the initial state, imply the desired or goal state.

The use of time parameters has great potential. With a properly extended semantics, it would be possible to express things which occur over time, and events which have duration. It is also possible to express concurrent action. This thesis does not discuss these matters in any detail.

Next, a deterministic variant of Rosenschein's bigress algorithm is presented which allows the synthesis of robot plans in a best-first manner. An implementation of this algorithm for the University of Alberta's Kato-Heron robot follows. This implementation uses a default rule of inference which, in effect, states that unless otherwise noted, everything remains the same. This is much more efficient than the frame axiom approach which requires an inference for everything which does not change.

The robot has not executed any plans generated by the planner because, to date, no one has programmed the middle level control routines for navigation and sensory perception. This is a major task made more difficult because of a lack of repeatability of the robot's motions and because of difficulties in interpreting the results of sonar scans.

The major result of this work is that it is not necessary to resort to a modal logic, but that it is possible to create a feasible implementation of a planner based on first order function free logic.

2. Some Robot Systems.

Historical Notes

For centuries man has dreamed of making machines in his own image. Around 200 BC, Heron of Alexandria created statues of gods that spoke, gestured, and even prophesied. He left a written account of how to construct them, making him the first authority on robotics. The robot at the U of A was named in his honour.

As the years rolled by, "automata" as ~~they~~ were then called, grew more and more complicated. A highly realistic clockwork doll produced by Jaquet Droz (Aleksander and Burnett, 1983) around 1775 could play a model piano, while another could write words. George Moore (Aleksander and Burnett, 1983), in 1893, built an "automaton" that could walk around freely. It was powered by a simple steam engine housed within the body. The exhaust was through a hollow cigar in the figure's mouth.

Of course these things were mere toys. Useful "robots" came into being with the Jaquard loom with punch card programming in the late 1700's. About a century later, the Warner and Swayze arithmatic turret lathe was developed. This machine had eight mathematically laid out cams to program (in the sense of a computer program) the cycle of feed stock handling and product machining. A similar machine of the same time period was the automatic milling machine which needed a human operator only for placing and replacing the metal stock.

However, these devices were not called robots; the word did not even exist before 1920, when Karel Capek in his play *Rossum's Universal Robots* used it to describe mechanical machines modelled on the pattern of human beings. These machines supposedly did not have any of the many human weaknesses, but eventually they turned on their human masters. The word itself was coined from the Czech word "robotnik", meaning serf.

There is no clear modern definition of a robot; just about everybody has a different idea of what can and cannot be so designated. The Robotics Institute of America define a robot to be a reprogrammable manipulator designed to move parts, tools, or special devices through variable programmed motions for the performance of a variety of different tasks.

This definition is not at all in the tradition of Capek. A better one is Nitzan's (1979) definition, "A robot is a general purpose machine that, like a human, can perform a variety of different tasks under conditions that may not be known a priori". This requires the robot to have a certain amount of inherent flexibility.

The lack of *a priori* knowledge also requires that the robot be able to gather information during execution. Thus a set of sensors, together with some method of understanding sensory input is essential in a robot; it must be a "closed loop" device. The first closed loop (or feedback) device was James Watt's (c. 1770) flyball "governor" for his steam engine. It would, however, be stretching things to call it a robot.

One of the first "sensory" robots was the turtle built by Greg Waller (Aleksander and Burnett, 1983). It was a small wheeled robot with a small photoelectric eye. The turtle could wander around, avoiding collisions with its "eye", until its battery power began to run down; then it would search for its hutch, and plug itself in to recharge its batteries.

2.1 Some Modern Sensory Robots.

There have been a huge number of modern robots with sensory feedback in the past few decades; some of the more notable have been Shakey, Jason, HILARE, and the JPL rover.

The Shakey project (Raphael 1976), developed between 1968 and 1973 at the Stanford Research Institute, was a pioneer in the field of "intelligent" robots. The fields of research encompassed by the Shakey project included pattern recognition, problem solving, knowledge representation, and natural language processing. It was equipped with proximity sensors and a TV camera, and was linked to a central computer by radio. Orders were fed to it by typing simple English into a terminal linked to the central computer.

Shakey spent most of its time in a world of seven rooms variously interconnected with 8 doors. These rooms contained several large boxes, and a few ramps. Shakey was able to push the boxes from room to room, and even stack them by pushing a box up a ramp placed next to another box. It did this in accordance with plans generated by its main planning program called STRIPS.

The JPL robot research program (Thompson, 1977) was aimed at making a working robot operate effectively in a real world environment with particular emphasis on the creation of an interplanetary robot. The robot had a manipulator arm, a laser telemeter, and a stereoscopic camera. It was designed to analyse the general and then the local scene for examination of possible samples, and to allow it to navigate appropriately in its environment. There were even plan generators to allow it to navigate according to orders sent from Earth.

It incorporated one particularly good idea; the use of "precalculated freeways" for moving the manipulator arm. This saved considerable computation, and guaranteed the avoidance of obstacles such as camera mounts on the body of the rover.

The JPL robot borrowed heavily from knowledge gained from the Shakey project. In fact, Charles Rosen of SRI referred to it as "Son of Shakey" (McCorduck, 1979). Although the robot had a large impact on the development of various unmanned spacecraft such as Voyager, and Viking, the program was unfortunately abandoned in 1980 due to fiscal restraint.

Jason (Smith and Coles, 1973; Coles, Robb, Sinclair, Smith, and Sobek 1975) developed at Berkeley between 1970 and 1975, was a relatively inexpensive robot with the ability to manipulate simple objects and navigate in a real world setting. Its basic purpose was to investigate how an inexpensive real world robot system might be designed, and what problems might be encountered. In this way, it was very similar to the U of A robot.

Jason's senses consisted of a TV camera, infrared and ultrasonic proximity sensors, together with contact feelers; it planned through a combination of a STRIPS-like mechanism and decision analysis. Jason demonstrated that an inexpensive robot could be built, and be useful for real research. It provided a testbed for two plan generators.

The French robot HILARE (Chatila, R. 1982; Giralt, G., Sobek, R., Chatila, R. 1979), inspired by Jason, is under continuing development at the Laboratory of Automatic Analysis of Systems, in Toulouse. The HILARE has three main component systems:

1. The locomotion system,
2. The computer system,
3. The perception system.

The locomotion system is microcomputer controlled and consists of two stepper motor controlled rear drive wheels, and a free castering front wheel.

The computer system has three levels:

1. On board microcomputers which drive the perception system and the locomotion system.
2. A dedicated mini-computer which does most of the data processing, in particular, handling tasks associated with the navigation of the robot.
3. An IBM 370/168 computer, not central to the robot, which intervenes for short periods during the functioning of the robot, i.e. during learning phases, or for elaboration of a

complex decision.

The perception system also has three sub systems.

1. A 3-d perception system which is able to find the position and orientation of observed obstacles, using a laser range finder for position and orientation, and a camera to recognize objects.
2. A tracking system used in a room delimited by beacons using infrared emitters.
3. An ultrasonic system which does obstacle avoidance and some navigation tasks.

HILARE forms plans with a production system, whose rules have two parts; the left hand side states what conditions must exist for the rule or action to fire, while the right hand side details the consequences of the rule.

The system searches for a plan by applying rules to the current state. These rules are activated by differences between a given goal and the current state. To this extent, it is very similar to the STRIPS paradigm, but it differs, in that the planner is nonlinear; its plan is merely a set of rules, ordered only by a set of plan constraints.

This allows parallel execution, i.e. several rules may be applied at once. In addition, this minimal ordering permits the plans to be in a very general form. Thus plans and plan fragments can be easily adapted, or reused under a variety of conditions.

The CMU rover is yet another mobile robot. Moravec (1984), one of its codevelopers, feels that the mobile robot is essential to the development of intelligent robots, because these must be able deal with unfamiliar terrain, thus requiring a high degree of flexibility.

The CMU rover forms plans by use of a system very similar to the Hearsay II speech understander (Erman, 1980). A number of independent processes operate within the robot simultaneously. They communicate via a "blackboard". For example, if the touch sensor's process reports an imminent collision to the blackboard, the collision avoidance process will speed up its execution by being given a higher priority.

Carnegie-Mellon University is a very active site, and has been responsible for the development of a number of other robots including the Neptune mobile robot (Podnar, Blackwell, and Dowling, 1984), the Terregator outdoors robot (Kanade, and Thorpe, 1985) and the new Uranus robot (Podnar, 1986). A field of particular relevance to the University of Alberta's robot project has been their work in sonar mapping (Elfes, 1987).

2.2 Description Of The Kato-Heron Robot

The robot is a heavily modified Heathkit ET-18 Hero robot. It is mobile and possesses an arm with 4 degrees of freedom. The robot's major sensory device is an ultrasonic sonar device although plans exist to mount a camera on it.

The basic robot came equipped with a 6808 microprocessor, a removable hex keypad, and a variety of motors and sensors. It has now been connected, via a 9600 baud serial link, to our Vax 11/780, so that its computational ability has been greatly enhanced. Our robot also came with the optional voice module, and the optional arm-gripper assembly.

There are three major parts to the robot: the chassis, the torso and the head. The chassis holds the batteries and the wheels; the torso contains the majority of the electronics; and the head contains the sensors and the arm.

Kato has a steerable front drive wheel, and two rear roadwheels. The robot can turn almost entirely within its own diameter. Some problems with the predictability of the robot's motion have been partially overcome by proper calibration and sensory feedback. Larger batteries have been installed, allowing extended periods of operation unconnected to the battery charger.

The head with its arm and sensors is the most used part of the robot. The arm is quite weak, capable of lifting only about a pound and able to grip easily only certain special shapes. There are numerous sensors, ranging from ultrasonic sonar to light detection.

Sense Specifications

Kato's major perceptual sensor is the ultrasonic range finder, which performs various navigational and obstacle detection functions. The maximum range of the sonar device is about 2.4 meters, and its resolution is 1 cm. The broadcast frequency is around 32 KHz. Beam width is advertised to be about 30 degrees vertically and horizontally. In reality, it is about 40 degrees. With a horn shaped sound cone attachment to the ultrasound broadcaster the beam width is narrowed to about 20 degrees. Kato's other sensing devices include a light sensor which provides an ambient light measurement, and a sound sensor which provides an ambient sound measurement in the range 200 to 5000 Hz.

In addition there exists a motion detector which can provide an interrupt to the CPU if a significant change in the ultrasonic echo pattern is detected; it will pick up an average sized adult walking toward the robot at a distance of about four meters.

There is a speech generator on the robot. It is not the one supplied by Heathkit, but the somewhat better SSI 263 chip. All development of the speech synthesis routines were carried out by Scott Stacey (1986). These routines have proven to be useful when debugging the robot, because the robot can explain its intended actions before it actually executes them.

Motor Specifications

The head of the robot rotates about 340 degrees. The robot's arm has four degrees of freedom. Including the rotation of the head and the main drive of the robot almost gives the arm the illusion of 6 degrees of freedom.

The shoulder motor raises and lowers the arm over 150 degrees of motion; the extend motor extends and retracts the gripper 5 inches; a wrist pivot motor pivots the gripper 180 degrees; the wrist rotate motor rotates the gripper 180 degrees; and the gripper motor opens the effector 9 centimeters. The maximum payload for the arm is 500 gm, and its gripping force at the tip of the parallel acting gripper is about 140 grams.

There are two micro-switch touch sensors on the gripper. One acts as a probe, informing the robot of the presence of an object. While the other indicates to the robot when the gripper has closed on an object.

The arm has now been relocated on the head to operate in the same general direction as the sonar. This facilitates measuring with the sonar and grasping with a minimum of head movement.

Unfortunately, there is only one motor controller for the robot head and arm, so that only one arm motor at a time can be activated. This makes complex movements of the arm well nigh impossible. It is possible, however, to move the drive wheel, the steering wheel and the head (or indeed any arm motor) simultaneously. To date, this has proved of limited use.

Projected Modifications

There are plans to add a video camera, and an on-board M68000 processor to do image analysis.

Software

There is a resident program on the M6808 processor on the robot. Its major functions are to control the robot motors and service the serial communications link. It is also capable of performing a sonar scan over a set of angles.

The Vax low level software is essentially a set of *C* routines. These are called by user programs, and provide the interface to the robot processor.

Some of these routines are:

1. Getmem, Putmem - to transfer data to and from the robot's memory.
2. Motor functions - to drive the various motors.
3. Cscan,scan- to perform an ultrasonic scan of environment.
4. Sense- to give reports from various sensory equipment.
5. Speak- to speak the designated words.

There are also some routines for our Sun workstations, which draw and maintain a graphic image of the robots' location, and various scan results. These have proven to be quite useful when trying to interpret incoming sonar images.

Currently, the robot is able to locate, close with, and pick up one of the sonar trees. These trees are objects specifically designed to be visible to the sonar and easily manipulated by the gripper. It then places the tree at a predesignated location. During execution of these actions the robot gives a running commentary with its speech synthesis mechanism. Concurrently, the Sun workstation graphically displays the robots location and the results of its sonar scans. The University has produced a videotape of the robot performing these actions.

All of this is accomplished by a program written in Franz lisp. In order to expand this program, or to get the robot to do even slightly different tasks requires a significant programming effort. The need for the robot to have more flexibility in its actions, without such great effort, was a major motivation for this planning project.

3. Robot Planning Systems.

Robots must perform a variety of tasks under conditions that are not entirely known beforehand. This typically requires that the robot have some way to decide what it should do, shortly before it does it; thus it needs some form of planning system.

Industrial robots rarely do any planning. Instead, in much the same manner as the current fixed programming of the Kato Heron robot, they do only as they are told, and invariably, a great deal of effort is expended in telling them exactly what should be done. There is a clear requirement for some form of flexible robot planner.

Robot Programming Languages

Industrial robots are traditionally programmed in one of two ways. They can be shown how to do the job, by using a simple teach and repeat system, or they can be programmed via a robot programming language. For a sensory robot, whose actions are not wholly predetermined a robot planning language is a necessity. For a detailed survey of robot planning languages see Gruver, Soroka, Craig, and Turner (1983), and for greater detail on what robot planning languages can't do see Soroka (1983).

Simple pushbutton teach and repeat schemes are adequate to program a robot for a wide variety of relatively simple tasks. For more complicated tasks such as picking up objects from a pallet, where each new position can be calculated from a start position and an increment measurement, a programming language is indispensable. A further advantage of an robot planning language is that it allows a robot to be programmed off-line. In fact, a proper programming system should integrate CAD/CAM systems so that the robot program is generated automatically as a by-product of the CAD process. Finally, provided sufficient accuracy between robots is available, textual robot programs have the potential to be portable,

Background of Robot Programming Languages

Robots during the 60's were programmed only in the teach and repeat mode. This was adequate for applications such as spray painting and spot welding where a single task was endlessly repeated. As both tasks and robots became more complicated, there arose the need for a robot planning language.

Stanford AI labs developed WAVE in 1973 as an experimental language for programming robots. [illegible], when hooked to to a vision system, demonstrated successful hand-eye coordination.

In 1974, the Stanford group began to develop an Algol based robot programming language, AL, incorporating constructs for controlling multiple arms in parallel and cooperative tasks. An interactive teaching module was added to an existing compiler, leading to the highly interactive system in use today.

Other institutes quickly followed with their own robot planning languages. Since 1976, IBM has been responsible for a number of languages including EMILY, ML, AUTOPASS, and AML. SRI International developed RPL, and the Jet Propulsion Laboratory wrote JARS in PASCAL. JARS was part of the NASA effort to control robots in the construction of solar cells. The first commercially available robot language was Unimation's VAL, in 1979. It is an extension of BASIC.

In 1981 Automatix developed RAIL to control visual inspection, and McDonnel Douglas developed APT. Two more came out in 1982, AML from IBM, and HELP from General Electric. Even more new languages are no doubt forthcoming.

Unfortunately today's robot programming languages are deficient in a variety of ways. The control level is typically separated from the user interface so that new algorithms are hard to test and debug. They achieve completeness in robot performance only by adding complexity. Communication with external devices is difficult, and they cannot adapt to changes in architecture such as adding an arm or adding an additional processor.

There are other problems with the current crop of robot planning languages. Many actions in a factory include two or more machines. Most current robot planning languages have great difficulty in coordinating several machines, even for the simplest of tasks, so that is very costly to program several robots with a robot planning language.

A final difficulty with current robot planning languages, is that they are installation specific. This means that the program that you slaved to write will be thrown away with next years crop of "improved" robots, or at the very least the user will be tied forever to one manufacturer's robot line. This portability issue is nothing new for computer languages.

What is needed is a task description language. This will prevent situations where the failure of one robot in a team of two robots, causes both robots to become idle, even if the working robot could reach the area accessed by its partner. In addition, task descriptions can allow a temporarily idle robot to assist the other robots around it.

Furthermore, the language must have clear notions of priority and interrupts, along with mechanisms for describing parallel activities. Things, such as the points at which a motion can be interrupted safely, and the relative priorities between two tasks must be specified. The exact decision of which arm does what task can then be left to an "intelligent" scheduling program.

Other important features are good knowledge representations, the ability to do parallel computing, good ways to describe obstacles, and hierarchical methods to distribute computing resources.

3.1 Planning systems

The better robot planning languages shift the burden away from the programmer and onto the robot. Thus the robot no longer executes an intricate series of instructions, but actually creates its own plans based on a given set of goals. This problem solving ability is the key to flexible robotics.

An influential early problem solving system was Newell, Shaw, and Simon's (1960) General Problem Solver (GPS). It was first conceived in 1957. Since that time, the program has existed in several versions, each of which was designed to face a slightly different set of difficulties arising in the construction of a general problem solver.

There are three parts to GPS. First is the problem formulation, followed by the problem solving methods. Finally, there is GPS's internal representation of the problem.

The problem formulation consists of an initial object (ie world state) and a set of desired objects (the goal states), together with a set of operators which will transform objects from one state to another.

The problem solving methods include a set of "differences" between various objects, and a difference ordering. This difference ordering is often the secret to success. Poor ordering will usually cause GPS to fail.

The basic heuristic used by GPS in guiding its search is means-ends analysis. First GPS finds the "biggest" difference between what is given and what is desired. It then goes about reducing the difference by applying operations onto the objects. A new object is thus generated. GPS now reduces the new object's differences.

GPS successfully solved a number of problems such as the Towers of Hanoi, and the missionaries and cannibals problem. However most of its cleverness was contained in its difference ordering. When an inappropriate difference ordering was given to GPS, it would usually be unable to solve the problem. For example, when dealing with the Towers of Hanoi problem, the differences between objects would include the location of each of the various disks. The location of the larger disks are more important than the location of the smaller disks, because once a larger disk is in location, it doesn't have to be moved to get a smaller disk in the correct location. This gives us our difference ordering.

Cordell Green (1969) was a pioneer in applying predicate logic to robot planning problems. He used axioms to describe the effects of actions upon state descriptions. Each state was specified by a set of formulae. To keep the formulae describing each state distinct, he included a state designator as a part of every proposition.

Problems were solved by giving to a resolution theorem prover a set of axioms describing the effects of actions together with a set of predicates describing the initial conditions. The theorem prover was then asked to prove the existence of a state where the given goal description was true. Since the method of proof was constructive, knowing that the state existed, also meant knowing how to achieve that state.

On(A,B,Do(stack(A,B,S0))) is a typical proposition. It translates roughly as given that you stack A on B in state S0, then there is a new state called Do(stack(A,B,S0)) where A is on B. A successful proof attempt would generate a plan expressed as a composition of Do functions. For example, if the next action was to stack block C on block D, one proposition of that state would be: On(A, B, Do(stack(C, D, Do(stack(A, B, S0))))).

Unfortunately, this system requires not only axioms which express the changes that a certain action will involve, but also requires axioms to express what hasn't changed after a certain action; these are called frame axioms. In a large system, there are a lot of facts which don't change after a certain action so that frame axioms impose a big burden upon the theorem prover. In addition, the amount of theorem proving search required for a problem solution expands explosively with the difficulty of the problem.

These two problems, the frame problem and the search problem, made Green's formulation unworkable for nontrivial problems, and were a major incentive to the development of other forms of robot planning.

One subsequent system was Fikes and Nilsson's (1971) influential STRIPS system. STRIPS was implemented on the Shakey robot, and did trojan service until 1975 when the Shakey project was closed.

STRIPS maintains a stack of goals, and focuses on solving the top goal in the stack. When the top goal is satisfied, it is removed from the stack, and the next goal is considered.

If the Top goal is a compound goal, then the compound is broken up, and its subject parts (sub-goals) are placed on top of the stack. Each part is then worked on until it is solved. When all of the parts are solved, the entire compound goal is reexamined. If it is all true then it is considered solved. If not (due to goal interaction) then the sub-goals are re-ordered, and the entire compound goal reconsidered.

All goals in STRIPS are statements about the state of the world. For example, regarding two blocks A, and B, a typical goal would be On(A,B). With the addition of an additional block C, another goal could be (On(A,B) & On(C,A)).

Changes in the state of the world are accomplished by "operators". Each operator has an "add list" and a "delete list". The add list is a list of statements about the world which become true when the operator is applied. Similarly the delete list removes statements which are no longer true, For example a simple operator might be Stack(A,B). The add list would contain On(A,B), and the delete list would contain Clear(B). In this way STRIPS sidesteps the frame problem. Everything that hasn't been deleted must still be true! This saves an enormous amount of computation.

Not all operators can be performed at all times. If block C was on block B, the operation Stack(A,B) could not be performed directly. Each operator, then, has a list of "preconditions" which must be true before the operator can be applied.

STRIPS resolves a goal literal by first attempting a proof attempt of the current goal clause against the current state. If the proof attempt succeeds then the current goal must be true, and it is removed from the goal stack. If it fails, it tries to deal with the unresolved literals (or differences) by trying to find an operator whose add list contains a literal which can be matched to the remaining goal literals. This operator is now added to the stack. On top of the operator is added its preconditions.

The preconditions are now considered goals. If they are already true, they are swiftly removed from the top of the stack, otherwise more operators, with more preconditions must be found.

When the top item of the stack is an operator, its preconditions must be true (since we already solved them above) and so the operator is applied, changing the state of the world. It is now removed from the stack, and STRIPS continues working on the rest of the stack until the stack is empty.

Using these techniques, STRIPS can solve the block stacking problem [ON(C,B) & ON(A,C)]. The easily obtained solution sequence is {unstack(c,a) stack(c,b) pickup(a) stack(a,c)}.

STRIPS does not do so well on the goal (on(b,c) & on(a,b)) given On(c,a). It gets a workable but overlong solution: {unstack(c,a) putdown(c) pickup(a) stack(a,b) unstack(a,b) putdown(a) pickup(b) stack(b,c) pickup(a) stack(a,b)}. Extra work is generated because STRIPS decided to do on(a,b) before achieving on(b,c). Goal interaction forced STRIPS to undo on(a,b) before it could achieve on(b,c).

A major problem with STRIPS is that it handles goal interaction in a very weak manner. It simply reorders the higher level goals, and tries again. RSTRIPS is an attempt to circumvent goal interaction by regressing conditions that would be clobbered by the achievement of another goal to before that goal is achieved. In this manner, it is usually possible to get the correct goal ordering with as little reshuffling as practicable.

Hacker (Sussman, 1975) was MIT's response to STRIPS. It would try to formulate plans to solve subgoals independently and then try to patch the plans together to solve the higher level goals. It used a library of bug correction procedures to patch the plans.

STRIPS, RSTRIPS, and Hacker are all one level planners. In other words, every step along the way is assumed to be as important as every other step, but this is not usually true. Many, if not most, of the steps in a plan are mere details while a few steps are vital to the eventual success to the plan. For example, when a person is planning to buy a new car, having enough money is usually considered more important than actually finding your way to the dealership. A planner should work on the more important parts first, and only consider the details later. This introduces the notion of hierarchy into planning.

A good early approach to hierarchical planning was ABSTRIPS (Sacerdoti 1974). The essence of this system was that "unimportant" details were ignored in order to create plans. Later these details would be considered in order to flesh out the plan.

In order to do this ABSTRIPS user identifies a hierarchy of conditions. The lower conditions are mere details, while the upper ones are the important factors in determining the plan. Determining which conditions are important is of crucial importance to the efficiency of the planning system.

ABSTRIPS works in a very straightforward manner. Simply solve the problem, in the manner of STRIPS, considering only the highest level conditions. This will generate a skeleton plan. It then uses this plan, together with the preconditions for the next lower level to generate a new goal stack, and again uses STRIPS on this goal stack to find a more detailed plan. This process continues level by level down to the lowest level of detail. If no solution at a lower level exists, then we must return to a higher level to find a new higher level plan to flesh out.

A fundamental drawback of STRIPS like systems is the need to choose an order in which to attack composite goals. This leads to problems when the wrong order is chosen, and goal interactions force backchaining, or sometimes even force failure. A superior approach is not to order the subgoals unless goal interaction problems dictate the ordering. This is called nonlinear planning and is Sacerdoti's (1977) NOAH (Nets Of Action Hierarchies) method.

The initial plan generated by NOAH does not specify the temporal ordering of the subgoals, but regards them as conjuncts to be achieved in parallel. As the plan develops, goals will be ordered only if there is a reason to make one first (due to some presumed goal interaction). With this technique, NOAH adds constraints to a partial plan, rather than rejecting incorrect assumptions. Sacerdoti called it the principle of least commitment.

NOAH creates a lattice structure to develop and represent the plan. This structure represents the minimum required orderings among the operators it selects. NOAH is hierarchical in that it constructs first a skeleton of the plan, and at successive stages it fills in more detail. Let us consider how NOAH would solve the problem of creating a stack of 3 blocks.

NOAH first divides the problem into two subgoals, one for the middle block to be on the bottom block, and one for the top block to be on the middle block. It decides that the STACK operator must be used. The preconditions for STACK must now be considered. Note that the two goals are considered independently.

NOAH employs a set of "critics" to examine the plan, and detect interaction among the subplans. Each critic is a sub-program that can make observations about a proposed plan. This is very much in the tradition of systems such as HACKER.

The critics show that the two stack operations must be ordered to prevent conflicts. Ie. the bottom two blocks must first be stacked before the top two blocks can be stacked. The plan in now reordered, and redundant preconditions are eliminated. Again we expand the plan and apply the critics, another reordering takes place, this time to ensure that the bottom block is clear at the start. Finally once preconditions are checked we arrive at the required plan, constructively, and without backtracking.

Tate's (1977) NONLIN planner, is an evolution from NOAH. It can generate plans from task descriptions given in the task formalism. The task formalism is a way to describe actions in a hierarchical fashion. NONLIN generates plans at greater and greater levels of detail, and produces a plan as a partially-ordered network of actions.

NONLIN is an improvement over NOAH in that it is able to keep track of previous choice points, and backtrack to them upon failure. An ordering decision made by NOAH is irrevocable, so that the search space is incomplete and some simple block pushing tasks are unachievable.

The simplified control structure used by NONLIN is similar to NOAH's. It first starts with a single node representing the task to be performed. The following cycle then takes place:

1. Expand a node in the network using expansion from an appropriate schema.
2. Correct for any interactions introduced.
3. Repeat from 1 until there are no further nodes to expand.

If step 2 determines that an ordering decision was a wrong one, it backs up the process to the alternate ordering. Tate shows that only one alternate ordering is needed for completeness in search.

Another modern system is Stanford's Molgen (Stefik 1981) which is a hierarchical planner to assist molecular geneticists in planning experiments. It uses constraint propagation to allow the deferral of ordering decisions as long as possible, even though the decisions interact with each other.

A simple example will illustrate the concept. Suppose we wish to purchase both a new sofa and a new chair for our living room. Using Molgen's process, we would break this into two separate goals: Purchase(sofa = x) and Purchase(chair = y). Now, the purchases are not really independent, after all both will impact our pocket book. Furthermore, if we buy a orange sofa then the chair probably should not be green. Thus we add some constraints to the global constraint list: Compatible(x, y) and perhaps Below$500(x, y).

The planning now proceeds independently, perhaps adding more constraints. At some point, it will become necessary to substitute some values for the variables involved. This involves a constraint satisfaction process which ensures that all of the posted constraints are satisfied before a substitution can be considered viable.

Deviser (Vere 1983) is a general purpose automated planner-scheduler to generate parallel plans to achieve goals with time constraints. It is on a direct evolutionary path from NOAH and NONLIN. Its major achievement is that it deals directly with time information, creating plans to achieve goals which may have time restrictions on when selected goals should be achieved and for how long goal conjunctions should be preserved.

The plans produced are a partially ordered network of activities. For each activity there is a duration and a start time "window" attached.

Deviser would chain backwards from unordered subgoals by satisfying goals where possible by linking goal nodes with the same, already achieved, nodes. If subgoals cannot be met by linking, nodes are expanded in parallel step by step, into activities which achieve the subgoals. When two parallel expansions produce contradictions, conflicts are resolved by ordering the nodes. If the conflict cannot be resolved by reordering them, Deviser backtracks to the last choice point and tries an alternative.

Recognizing that actions and events have deterministic durations, and scheduled events may occur over which the actor has no control was a major step forward in planning. Deviser's main motivation for dealing with time was the intended application to unmanned spacecraft such as Voyager.

3.2 The need for Logic

All planning systems have to have knowledge about their environment. Furthermore, it is unlikely that all of a planners needed information can be placed into a machine in immediately accessible form. Therefore, there is a requirement for some sort of inference mechanism so that information implicitly present can be deduced. And indeed, most of the above planners use a form of logical deduction to deduce information about their worlds.

It was soon noted, that planning process itself could be cast as a deductive problem, and therefore planning could be done by logical inference.

There was a small problem. Traditional logics are monotonic; the number of statements known to be true is strictly increasing as "facts" accumulate, i.e. a statement known to be true can never be made false. This made straightforward application of traditional logics difficult because facts and axioms are inaccurate, and therefore conclusions must be tentative.

Rich (1983) says that monotonic systems are not suitable for dealing with three kinds of situations that can often arise in real problem domains: incomplete information, changing situations, and the generation of assumptions in the process of solving complex problems. These factors have led to the development of nonmonotonic systems.

Doyles (1979) Truth Maintenance System (TMS) is an example of a system which supports non-monotonic reasoning. Its role is to maintain consistency among a database of statements generated by some other system. When an inconsistency is found it invokes a mechanism to resolve the conflict by altering a minimum set of beliefs.

Each statement (or node) in TMS is in one of two states: IN or OUT. A node is IN if it is believed to be true. A node is OUT if there are no valid reasons for it to be believed true. In addition, each node has a list of justifications, any one of which could establish the validity of the statement.

A simple example will illustrate how TMS works. Statement (1) "it is cold" is believed to be true, based on the justification that statement (2) "it is winter" is also believed to be true. If at a later time (2) is no longer believed to be true, then (1) would automatically be placed on the OUT list of nodes.

Monotonic systems however, have a great advantage over nonmonotonic systems: no checks are needed to see if there are inconsistencies generated between old knowledge and new statements.

Dynamic logic is a monotonic system useful for the statement of robot planning problems. It was originally developed to prove program correctness (Litvintchouk and Pratt 1977) and program correctness and correct planning are strongly linked. After all, what is a program, but a plan for a computer?

In fact, Rosenschein (1981) has formulated the planning problem in terms of propositional dynamic logic. In addition, Rosenschein gives a bidirectional algorithm (bigress) which can plan over a broad set of plans dealing with conjunctive and disjunctive goals, and nondeterministic actions. In this system, a plan is considered correct only if it can be proven so in a formal language.

Kautz (1982) has extended Rosenschein's work and defined a first order dynamic logical planning language which has extended bigress to cover a subset of this language.

Dynamic logic is a modal logic: a plan is a reachability relationship over a set of possible worlds. When B is a plan, [B] p is true in a world I if p is true in every world reachable from I by B. So, [B] p can be read as "after B, p."

The syntax of Kautz's dynamic logic is similar to function free first-order logic. An "action symbol" applied to a sequence of terms is the simplest kind of plan which can appear inside a []. Complex plans are built up by sequencing [A;B] and alternation [P==>A,B].

The semantics of his dynamic logic includes a domain of individuals, a set of worlds, and an interpretation for the action symbols. Each world interprets the terms as members of the domain, and the predicate symbols as predicates over the domain. The meaning of the plan is a binary relationship over the set of worlds.

Both Rosenschein and Kautz define the planning problem as a triple (V,Q,R(u)), where V is the vocabulary of the problem, Q is the set of domain constraints, and R(u) is a finite set of statements called plan constraints. These plan constraints are a finite set of wffs of the form p ⇒ [u]q. A solution to the planning problem is a plan A such that it is provable from the domain constraints Q which satisfies all of the plan constraints (i.e. Q|– p ⇒ [A]q)

Bigress, a nondeterministic algorithm, starts two searches, one starting at the initial state, and the other at the goal state. It works by chaining backwards (or regressing) from the postconditions of actions, and forward (or progressing) from preconditions of actions, until the two searches intersect. It then returns a list of actions needed to go from the start state to the goal state.

If a conditional occurs, then bigress starts two forward searches, one assuming that the conditional is true, and the other assuming the conditional is false.

When Rosenschein talks about progressing or regressing an action he really wants to discover the strongest provable postcondition or the weakest provable precondition, respectively. Given domain constraints Q, the strongest provable postcondition of a nonmodal wff r and an action A is defined to be a wff, called r/A, such that $Q \vdash r \rightarrow [A]\, r/A$ and $Q \vdash r/A \rightarrow q$ whenever $Q \vdash r \rightarrow [A]\, q$. Similarly, the weakest provable precondition of a nonmodal wff s and an action A is a nonmodal formula A\s such that $Q \vdash A\backslash s \rightarrow [A]s$ and $Q \vdash q \rightarrow A\backslash s$ whenever $Q \vdash q \rightarrow [A]s$.

Thus, in essence, progressing an action means to derive, from a given world description and a given action, a new world description that accurately (and completely) depicts the state of the world if that action were performed, and to regress an action from a set of goals means to derive a new (minimal) set of preconditions (or goals) that must occur so that the given action, when performed, will achieve the given set of goals.

Note that it would be more desirable to have the strongest postcondition and the weakest precondition instead of the strongest provable postcondition and weakest provable precondition. This is impossible to express in a nonmodal formula because of the monotonic nature of the logic, and because of the way that the axioms are used to characterize actions. Axioms can always be added to the domain which would strengthen postconditions or weaken preconditions and so we must settle for the strongest provable postcondition.

The weakest provable precondition a\q of an action a over a condition q is found by taking the disjunction of the set of formulas, each of which is a conjunction of a set of Pi drawn from the "left hand side" of the dynamic axioms of Q (i.e. Pi → [a]q) such that conjunction of the corresponding qi's implies q.

Let us look at Rosenschein's sample axioms:

1. A → [a] (B v C)
2. G → [a] ¬B
3. (F & E) → [a] D

In this case, a\(C v D) = (A & G) v (F & E). This is because (B v C) conjoined with ¬B implies (C v D), so the conjunction of the corresponding left-hand sides (A & G) is one disjunct of a\(C v D). Likewise, the formula D alone implies (C v D), making the corresponding left-hand side (F & E) the second disjunct. These two cases are the only ways to obtain (C v D).

The method for obtaining p/a is the dual of a\q, where we examine the "right-hand side" of the dynamic axioms.

Rosenschein does not provide a proof that his method obtains the weakest provable precondition.

Kautz, however, shows that, given the correctness of progress, regress, and derivability, the bigress algorithm is correct. He also shows that, given the correctness and completeness of progress, regress, and derivability, the bigress algorithm is complete. He goes on to say that in certain pathological cases progress and regress are not complete because the strongest provable postcondition (weakest provable precondition) cannot be represented by a finite length nonmodal formula.

Kautz presents an example of how bigress solves a variant of the register swapping problem. This problem has proven difficult for STRIPS and its descendants.

There are several boxes, B1, B2, B3, each of which can hold various coloured stones S1, ..., S10. The dump action, Dump(B1,B2), transfers all the stones from one box to another.

Static axioms such as Box(B1), and Stone(S3) describe basic facts, while the dynamic axioms describe (perhaps only partially) the effects of the action Dump.

1. (All x,y,z) ((in(x,y) & box(z)) => [dump(y,z)] in(x,z))
2. (All x,y,z,w) ((in(x,y) & ¬(w = z)) => [dump(y,z)] in(x,w))
3. (All x,y,z,w) (¬color(w,x) => [dump(y,z)] ¬color(w,x))

The last two axioms describe conditions which are invariant under an action, i.e. a frame axiom. STRIPS avoided explicit frame axioms because it maintained a single world model with the understanding that actions not explicitly removed from the database remain true after the action.

For many applications, this is quite efficient; however Waldinger (1977) states that for an action such as dump, which can affect an arbitrarily large number of objects, the STRIPS approach can be very wasteful.

Now that we have a vocabulary and some domain axioms, we can formulate the register exchange problem as follows:

(in(S1,B1) & in(S2,B2) -> [u](in(S1,B2) & in(S2,B1))

A solution u = {dump(B1,B3), dump(B2,B1), dump(B3,B2)} is found with no backtracking. This is because the backwards search found only dump(B3,B2) and dump(B3,B1) as possible last actions. There is no other possible last action.

Linear planners such as STRIPS fail on this problem because an attempt to fulfill one part of the conjunction directly causes the other conjunct to fail irrevocably. NOAH first tries to create independent subplans, finds that its subplans cannot be combined, and is forced to replan. RSTRIPS solves the problem in a manner very similar to this application of dynamic logic.

Note that the dynamic logic approach is different from the STRIPS paradigm. In STRIPS, actions are not regarded as mappings from states to states but are syntactic transformations of state descriptions to other state descriptions. This leads to the great advantage of not having to mention the things that do not change (i.e. the frame conditions). This also leads to the disadvantage of having to keep the transformations fairly simple (eg., addlists and deletelists).

A simple example of this is the toggle action, described by the pair of axioms:

1. $On(light) \rightarrow [toggle(switch)] \neg On(light)$
2. $\neg Onlight \rightarrow [toggle(switch)] On(light)$

This is almost impossible to specify with an addlist/deletelist pair because it cannot be determined in isolation whether toggle adds or deletes the formula On(light). With the toggle action, the consequences depend conditionally on the antecedents.

Another example of the flexibility of this approach is the three socks problem. How do we retrieve a matching pair of socks from our dresser in a dark room? Whenever we pick a sock we either get a black or a white sock. This is, of course, a nondeterministic action. I will not go into the details of the axiomatization but suffice it to say that bigress grinds out the solution: pick 3 socks from the dresser.

All of the other systems I have discussed would fail with this problem. Most of them cannot handle "picking up a random sock." The others (such as NOAH and Deviser) would try and deal with the problem by making two subplans, one to obtain a pair of white socks, and the other to obtain a pair of black socks. Neither subplan can be realized.

The use of a set of axioms to characterize the consequences of actions does allow great flexibility. However, in fairness to STRIPS, it should also be noted that this flexibility is achieved by using a large number of frame axioms for each action, and that these axioms have a huge computational overhead.

There is another problem. The length of a search is exponential w.r.t. the number of steps in the solution. Furthermore, it is likely that there will be a large number of possible actions at each step in the search for a plan. Therefore, the combinatorial explosion of the search is very rapid indeed. If we wish to retain the flexibility, and limit the size of the search we must prune the search tree severely at each step, and we must reduce the length of the search. The first can be done by careful choice of actions, and the second by hierarchical methods.

Rosenschein defines a hierarchical planning problem as a tree of single level planning problems. Higher level domain dynamic axioms are simply taken as low level plan constraints. In other words, for each higher primitive aotion, there is a lower level problem whose actions are more primitive than those of the higher problem. For example, the dump operator might be accomplished through a combination of grab, goto, and drop primitives.

4. A First Order Logical Approach to Robot Planning

There are some problems to a straightforward implementation of Rosenschein's dynamic logic approach. The nondeterministic nature of his algorithm makes it difficult to implement on a deterministic, serial machine. The frame axioms will impose a huge computational cost on any theorem prover, and besides, it has proven difficult to implement a good theorem prover for dynamic logic. Furthermore, there is no need to resort to the power of a modal logic, when normal predicate calculus will suffice.

The nondeterministic bigress algorithm allows Rosenschein to sidestep the issues of what states and actions to consider, and in what order. These have to be met when the algorithm is implemented on a serial machine. In the present implementation a number of heuristic functions have been added which allow the search for a solution to the planning problem to be conducted in a best-first manner. Because of the added complexity of adding assumptions to this heuristic search, the algorithm will not create conditional plans.

As noted above, the frame problem has been a major impediment to the purely logical planners (i.e. those using frame axioms.) My system will avoid many of these problems by the use of a heuristic default rule of inference when progressing or regressing actions. This default rule essentially says that unless otherwise noted, everything remains the same.

Of course, there are no guarantees that the rule will be correct, but then in a frame axiom approach there are no guarantees that the axiomatization will be correct.

The biggest motivation to abandon the dynamic logic approach is that there is no need to resort to a modal logic when it is not required. In normal clause form logic it is possible to formulate the precondition, action, goal process as a clause of the form:

$\neg \text{precondition}_1 \vee \neg \text{action}_1 \vee \text{goal}$

This translates as either the preconditions are not true, or we do not perform the action, or the goal conditions must hold. If we have another clause of the form:

$\neg$precondition$_2$ v $\neg$action$_2$ v precondition$_1$

we can build up a sequence of actions (ie. a plan) thus:

$\neg$precondition$_2$ v $\neg$action$_2$ v $\neg$action$_1$ v goal

If we happen to know that precondition$_2$ is true, we can then conclude that either we do not perform the plan or the goal will become true.

There is a small problem to the above formulation. It is logically the same as the formula:

$\neg$precondition$_2$ v $\neg$action$_1$ v $\neg$action$_2$ v goal

thus we need some form of action ordering. Dynamic logic avoided this problem because [a;b] is not the same as [b;a]. Green avoided this problem because Do(action$_1$(Do(action$_2$(S0)))) is not the same as Do(action$_2$(Do(action$_1$(S0)))). This problem can also be avoided with the use of ordered time arguments. Action$_1$(T_1, T_2) is defined to happen before action$_2$(T_2,T_3) because T_2 is in the second and first argument locations, respectively. This clearly is a syntactic notion of time ordering.

There is another, potentially huge, advantage to the use of time arguments. Given that we have a clear semantic notion of times and durations, this format can be used to easily represent concurrent or overlapping actions, or to represent scheduled events over which the planner has no control. The mechanism by which times and durations would be represented in dynamic logic or Green's formulation is not at all clear. Currently, in an attempt to keep things simple, the planner does not have the capability to express overlapping actions, but instead relies on the above simple syntactic method.

4.1 The Planning Language

The planning language is a restriction of function-free first order logic, with no existential quantifiers lying within the scopes of universal quantifiers.

Syntax

The syntax of the language is normal function-free first order logic in clause form (ie., with quantifiers eliminated and variables interpreted as universally quantified variables.) Variables will usually be denoted by lower case letters or words with or without numeric subscripts. They may also be preceded by a "?". eg. locationx,y_{13},?y_2, .. Individual constants will usually be words starting with a capital letter, with or without numeric subscripts (eg. Corner_i.) Skolem constants are capital letters with numeric subscripts (eg. D_1.) Skolem constants are turned into variables when we form the denial of the goal state.

Atomic actions and atomic propositions are defined in the usual syntactic manner. Atomic propositions are in infix form, while atomic actions are in prefix form. This greatly assists the readability of the language. By convention, the last argument of an atomic proposition, and the last two arguments of an atomic action are called time arguments. A typical formula is:

$\neg$(Kato isat Corner_1 Ti) v (goto Corner_1 ?locationx Ti ?Ty)

This translates roughly to "If Kato is in Corner_1 at time Ti then Kato went from Corner_1 to some unspecified location (?locationx) from time Ti to some unspecified time (?Ty)." The "isat" literal is an atomic proposition, while the "goto" literal is an atomic action.

Semantics

The semantics of the language are those of normal first order predicate calculus. Informally, we talk of things having a time ordering. This is established by having a syntactically restrictive definition of a plan.

If we were to relax the definition of the plan to allow, say concurrent actions, we would have to replace the normal semantics with some special purpose semantics to ensure that the time ordering over various actions and propositions remains valid.

Rules of Inference

The rules of inference consist of resolution, paramodulation, and factoring. Resolution means that when you have two clauses, P v R and ¬P v S, you can deduce the clause R v S. Factoring allows you to deduce Q v R from Q v Q v R, and paramodulation is a form of generalized substitution of equals. In essence, if we know a = b then from any clause Q having one or more instances of a (or b) we can deduce a clause Qa/b with one or more instances of a replaced by b.

These rules of inference are known to form a complete logic, in the sense that any valid formula can be shown to be valid. See Nilsson (1971) for a proof of this property.

4.2 The Planning Proolem

In general, the planning problem consists of a vocabulary, a knowledge base, some initial conditions, and some goal conditions. The vocabulary is a particular instantiation of the syntax of the planning language.

The goal conditions consist of a set of clauses describing what must be true at some goal time occurring at or later than the initial time.

The initial conditions describe the initial configuration of the world.

The knowledge base consists of a set of axioms which describe the environment of the planner, and of a set of heuristic functions, called by the algorithm, which perform a variety of functions peculiar to the world being modeled.

A plan is a collection of actions ($a_1, \ldots, a_n$), whose arguments are constants, arranged sequentially in time. Syntactically, this occurs when the second time argument of action a_i is the first time argument of action a_{i+1}. A solution to the planning problem (Voc, KB, Init, Goal) is a plan $P = (a'_1, \ldots, a_n)$ such that:

$$KB \vdash (Init \; \& \; P) \rightarrow Goal$$

where $KB \vdash A$ means that A is a logical consequence of the KB. This is the same thing as derivability since this logic is complete.

At the start, we know only the initial and goal states. A reasonable way to develop a plan is to start from the known states and to guess actions likely to lead to a solution. These actions [illegible] lead to the creation of new state descriptions, either by progressing them from the initial [illegible] (and its successors) or by regressing them from the goal state (and its successors).

Remember that progressing an action is to derive from a given world description and a given action a world description that accurately depicts the consequences of that action, and to regress an action from a set of goals and a given action is to derive a set of necessary preconditions so that the given action, when performed, will achieve the given set of goals.

This progressing and regressing will develop two tree shaped chains of state descriptions, one developing forward from the initial state, and one chaining backwards from the goal state. Each state description will be linked by an action and an implication to its ancestor. In other words, for each state Si and action Ai we can either progress it to obtain Si+1 and Si & Ai → Si+1, or regress it to obtain Si+1 & Ai → Si. If two states Si and Sj, Si developed from the initial state and Sj regressed from the goal state, are ever found such that Si → Sj then the search trees are linked. The postulated actions associated with the states on the path from the initial state to the goal state form the successful plan. This is the basis of my algorithm.

This algorithm is very similar to Rosenschein's bigress algorithm. Its major differences are that my algorithm is deterministic, and does not use modal logic. Furthermore, the algorithm does not allow the statement of conditional plans.

4.3 The Algorithm

The algorithm is essentially a best-first bidirectional search, forward in time from the initial state and backward from the goal state. Input to the algorithm consists of three parameters: the knowledge base (KB), the initial conditions, and the goal conditions. Output is either a plan, expressed as a sequence of actions ordered in time, or a report of failure when all possibilities are exhausted.

As there will be a large (possibly infinite, depending on how progression and regression are implemented) computational overhead involved in exhausting the search, the algorithm should have a higher level monitor to stop it when the search has gone on too long.

The KB has two components. The first of which consists of a set of axioms which describe the way the world works. This includes general facts about and properties of the world, but does not include the initial conditions.

The second component of the KB consists of a set of heuristic functions. These are used to determine how "reachable" one state is from another, and how difficult certain actions are to perform. Their purpose is to guide the bidirectional search. In addition, the KB contains the functions which allow us to proceed from one world state to another.

The initial state describes the configuration of the world before any actions are postulated. In particular, it is a set of clauses, all of whose literals, are dependant on a particular start time (usually time Ti). Furthermore, it consists of clauses likely to be needed in progressing actions, and hence in some sense, the "kind" of clause likely to be found in other; following states. In a sense, only the "changeable" clauses will be in the initial conditions.

The exact division between initial conditions and the knowledge base is not always clear. They have only been separated because a robot planner will solve problems over a wide variety of initial conditions, but typically will only have to face one set of universal laws. A brief example will illustrate. A typical train passenger has no control over what time a train leaves the station. Thus he would regard the current train schedule as part of the KB. A rail baron who could easily change the train schedule (although it might cost him some money) would more likely regard the schedule as part of the initial conditions, something to change if it got in the way.

The practical test between initial conditions and KB is simple. Do you want a clause to be considered at every proof attempt? If so, it belongs in the KB. Otherwise, it belongs in the initial conditions. There are no theoretical penalties to putting a clause in the wrong forum. If a clause is in the KB and really only applies to initial state then that clause will never be used apart from reasoning with the initial state because the time arguments will not match. If a clause is in the initial state, but should really should be in the KB, it again makes no difference, because we require that our progression method move all needed information from one state to the next. Therefore, the information contained in the clause, wrongfully placed in the initial conditions, will be progressed from state to state, and will be present in all proof attempts.

The goal state describes what must be true at the completion of the plan. It typically consists of a set of clauses dependant on some skolem constant time Tg.

The algorithm uses three lists: progress-list, regress-list, and choice-list. Progress-list contains the initial state and all of the states derived by postulating actions forward from the initial state. Regress-list contains the goal node and the nodes generated from the goal state. Choice-list is an ordered list of world descriptions from which we can postulate actions. The first element of choice-list is always the one believed most likely to lead to a solution. At the start of the algorithm, choice-list consists of the goal and initial states, in that order.

If choice-list is ever empty then there are no more states from which we can postulate an action. Thus the search has been exhausted and the algorithm fails. Otherwise, we choose the first element of choice-list, and call it "best-bet".

Assume that best-bet is also in regress-list (it must be in either regress-list or progress-list). We test to see if any node in progress-list implies best-bet. If one does then we have connected a node from the progress search tree to a node in regress search tree. Getplan assembles the plan by concatenating the series of actions needed to go from the initial node to the node in progress list to the series of actions required to go from best-bet to the goal node. The plan is returned and the algorithm terminates.

Only one connection will ever be found. The rest of the time we will be dealing with a large number of unsuccessful proof attempts. This can be expensive. One way to reduce the computational cost is to partition progress-list by the presence or absence of a number of features inherent in each state. A proof attempt is only tried when all features match. A typical feature could be the location of the robot. This heuristic device can ensure that large numbers of obviously unsuccessful proof attempts are not tried.

We now intensify our inspection of best-bet. If this is the first time that this node has been selected as best-bet, our heuristics will create an ordered list of what actions are possible from this state.

We remove the first action on best-bet's actions to be tried list. If there are no actions to be removed then best-bet has exhausted its possible actions, and is removed from choice-list.

Otherwise this action is placed on best-bet's actions already tried list and the utility of best-bet is re-evaluated.

Regress now uses the axioms associated with that action to generate a new state description from best-bet. This process is very similar to Kautz's (1982) weakest provable precondition method. This new description details what conditions must have existed prior to the chosen action achieving the state called best-bet.

This new state is fed into the theorem prover and "fleshed out" by applying the KB fact axioms to it. This is an attempt to bring to the surface any hidden relationships, which might be useful for further progression or regression. If the state contains a tautology (a clause containing two literals which resolve against each other) then the new state is discarded.

It is also heuristically evaluated to see how promising a node it is. This is done by attempting to prove that the initial state implies the new state. The unresolvable literals are weighted and summed to give us our distance measure for the new node. To this sum is added the cost of getting from the goal-node to the new node. This represents the heuristic cost of a node.

Finally, we re-sort choice-list, now containing the new state and a re-evaluated best-bet, and go to the top of the algorithm's loop.

The algorithm works in a similar manner when best-bet is in progress-list.

The Algorithm

```
Bigress(goal-node, initial-node, KB)
  negate(goal-node)
  progress-list :- (initial-node) ; regress-list :- (goal-node)
  choice-list :-  (goal-node initial-node)
Top-of-Loop
  If choice-list is empty then RETURN(Failure- no more choices)
  best-bet :- first element of choice-list
  If best-bet is in regress-list then
     if KB|— pre→ best-bet for some pre in progress-list then
          RETURN(get-plan(pre, best-bet))
     else
          choose a possible last action A that could achieve
          best-bet. Re-evaluate best-bet.
          if no action A available then
               remove best-bet from choice-list.
          else
               regress A from best-bet to create a new-node.
               expand new-node by applying KB fact axioms
               If new-node could exist then
                    evaluate new-node & append to choice-list.
                    append new-node to regress-list.
```

```
else (* best-bet is in progress-list *)
  If KB|— best-bet → post for some post on regress-list
    RETURN(get-plan(best-bet, post))
  else
    choose a "do-able" action A from best-bet.
    ie. one whose preconditions are entailed by best-bet.
    re-evaluate best-bet.
    if no A available, remove best-bet from choice-list.
    else
      progress A from best-bet to create a new-node
      expand new-node by applying KB fact axioms.
      Evaluate new-node and append to choice-list.
      append new-node to progress-list.
 re-sort(choice-list)
End-Of-Loop
```

Correctness of the Algorithm

Given the correctness of progress and regress, the algorithm is provably correct. In other words, any plan P = {A1 .. An} generated by the algorithm has the property:

KB|—(init & P) ➔ Goal.

Proof:

The algorithm terminates when KB|— Sj ➔ Sj+1 for some Sj in the progress-list and some Sj+1 in the regress-list. Sj was created as a series of progressions, and has a series of ancestors {initial, S0, ..., Sj}. In addition, each state Si has an associated action that created Si by progressing that action from state Si-1 (Si's immediate ancestor). In other words, the initial state via some action A1 was progressed to state S1. S1 was progressed via some action A2 to some state S2, and so on. Finally some state Sj-1 via some action Aj was progressed to the state Sj. The actions {A1 .. Aj} form the first part of the plan returned by the algorithm. By the correctness of progress we have:

KB|— Init & A1 ➔ S1

KB|— S1 & A2 ➔ S2

...

KB|— Sj-1 & Aj ➔ Sj

This can be rearranged to:

KB|— Init & A1 & .. & Aj ➔ Sj

Similarly, by the correctness of regress, we can show:

KB|— Sj+1 & Aj+1 & .. & An ➔ Goal

where the actions {Aj+1 .. An} form the second part of the plan returned by the algorithm.

Since $S_j \rightarrow S_{j+1}$ we see that:

KB|– Init & A1 & .. & An $\rightarrow$ Goal

Q.E.D.

Completeness of a Breadth-First Version of the Algorithm

Given the correctness and completeness of KB|– (derivability) and of progress and regress, and given that there can only be a finite number of possible actions at any one time then the algorithm can be made to be complete. In other words, given the above, if there exists a solution to the planning problem then it will be found.

Proof:

Assume that the successful plan has n actions. Engineer the planner's heuristics so that it conducts a breadth-first, left to right progressive only search. At each node, the planner will generate all possible actions, and then progress them into new states. It will then progress actions for all of the node's sister states. The planner will thus conduct an exhaustive search of all plans of length n before it looks at plans of length n+1. As there are only a finite number of plans, our desired plan will eventually be tested. This plan will be returned because the algorithm is correct.

Q.E.D.

There are very few practical advantages in making the algorithm complete in this manner. Furthermore, the theorem prover might not stop because of cycles (Lewis, 1975) in the deductive process. Thus, depending on how progress and regress are implemented, they may not be complete in certain pathological cases, because the weakest provable precondition (strongest provable postcondition) might not be realizable in a finite form.

A simple example will illustrate. If we have the following axioms in the KB:

1. $\neg$(?x on ?y ?tx) v $\neg$(?y above ?z ?tx) v (?x above ?z ?tx)
2. $\neg$(?x on ?y ?tx) v (?x above ?y ?tx)

the theorem prover can deduce an arbitrarily long clause of the form:

$\neg$(?x_1 on ?x_2 ?tx) v $\neg$(?x_2 on ?x_3 ?tx) v .. v $\neg$(?x_{n-1} on ?x_n ?tx) v (?x_1 above ?x_n ?tx)

In this pathological case, the theorem prover will not stop, and hence, progress and regress may not be complete. I can correct this by forking off a process every time the theorem prover is invoked. This process would assume that the prover had failed in its proof attempt. Thus, if the original proof attempt runs on forever this is equivalent to having a failed proof attempt.

This parallel version of bigress assures me that I will find a plan in all universes with finite possible actions. This assurance is still not of much practical advantage, as I am still resorting to an exhaustive search.

4.4 Support Functions Required by the Algorithm

A variety of support functions are used by the algorithm. Some are independent of the individual KB, and some are part of the KB. The most important group of functions independent of the KB is the set of functions which make up the theorem prover.

For a description of how these functions can be implemented see chapter 5.

The Theorem Prover

A theorem prover must satisfy two requirements of the algorithm. It must be able to determine whether one state implies another. Secondly, it must be able to expand a world description adequately to ensure that the level of detail is sufficient to allow the progress and regress actions full scope. This is done by applying the axioms of the KB to the current world description.

Other Functions Outside the Knowledge Base

The other functions found within the algorithm but outside of the KB consist of getplan, and makenode. Getplan is given two nodes, one on progress-list and one on regress-list, and returns the plan by concatenating the list of actions generated by looking at the ancestors of each of the nodes. Makenode takes a variety of properties, and places them on the property list of the name of a particular node.

Functions found in the Knowledge Base

There are eight functions to be found inside the KB. These include choosing actions to be progressed or regressed, progressing and regressing those actions, evaluating and re-evaluating world states, and determining which states could possibly imply one another.

For a description of one particular implementation of KB functions see chapter 5.

Choosing Actions for Progression and Regression

The ability to choose appropriate actions, and then be able to progress, or regress them from given world states are essential functions of the planning process. If these things are done in an inefficient manner, then the entire planning system will be inefficient.

Therefore, I have decided to place these functions into the Knowledge Base. This will allow various special purpose or domain specific inference techniques to be used, and will increase the usability of the planner in any specific world domain. In particular, it will allow us to use a variety of heuristic methods to minimize the effects of the frame problem.

There are however certain universal constraints on these functions. In the first place, the functions implicitly assume that the KB and the initial state contain all of the information needed to know about what actions are possible, including complete knowledge about that collection of clauses describing what must be true before that action, if performed, would be successful. This set of clauses is known as the preconditions of an action. In other words, any "do-able" action can be shown to be "do-able" because all of its preconditions are known to be true. For example, the robot needs to know that its hand is empty before it can pick up a sonar tree; this knowledge must be supplied before the choose-progress-action function can be expected to generate a pickup action as a possible next action.

The functions further assume that the progression and regression axioms (or whatever method is used in progressing and regressing actions) are complete in that any state developed from the initial conditions contains all of the knowledge that the planner needs to know about the preconditions of any action that might be postulated. In other words, like the initial state, any state derived from it must also have all of the "need to know" knowledge. The minimum requirements for an action to be progressable from a given world state is that all preconditions of that action must be true.

The situation when seeking actions to regress is not quite a mirror image of the progress case. There is no requirement for a certain "richness" of detail as the goal statement gives us the minimum requirement for success.

We must seek an action from which there exists a set of clauses which together with that action will achieve all of the goals. Thus, we must try to regress the action before we will know if it is "regressable". It is only when we regress the action and discover that it leads to an inconsistent state that it can be ruled out. Still we do not want to postulate every action as a possible one.

When deciding whether an action is suitable for regression, two questions should be answered. Does at least one postcondition achieve part of the current state? This question, although not logically required, ensures that the action accomplishes something. It also means that a lot of irrelevant actions will never be considered. Second, are any of the postconditions of that action explicitly denied by that action? This ensures that any obviously inconsistent actions are not postulated.

Progressing an Action

When the action has been chosen by the appropriate methods and heuristics the algorithm will try to progress it through the current state. Any method used must be sound, ie., it must satisfy the correctness requirements, and it must ensure that the knowledge of preconditions is passed on.

I describe here a logical method, adapted from Kautz, which will work [illegible] effects of an action are described by two sets of axioms. The action axioms detail the [illegible] action; and the frame axioms detail what does not change when an action is pe[illegible]

1. Instantiate all of the action's associated axioms (including the frame actions) with the particular form of the action.
2. Take the current state description together with the instantiated axioms and give it to the theorem prover with the current time already unified against the start time of the action.
3. Whenever the theorem prover deduces a clause which contains no propositions containing the current time, write it out. Note that the clause will always contain actions containing the current time. When the theorem prover stops, the new world description is complete.

Lets look at an example. We have chosen the action (pickup Tree_1 Corner_1 Ti ?tx) The associated instantiated action axiom is:

$\neg$(Kato isat Corner_1 Ti) v $\neg$(Kato hasfreehands Ti) v $\neg$(Tree_1 isat Corner_1 Ti) v

$\neg$(Tree_1 accessible Ti) v $\neg$(pickup Tree_1 Corner_1 Ti ?tx)) v (Kato isholding Tree_1 ?tx)

The associated frame axioms are:

$\neg$(x isat y Ti) v $\neg$(pickup Tree_1 Corner_1 Ti ?tx) v (x isat y ?tx) v (x = Tree_1)

$\neg$(x isbehind Tree_1 Ti) v $\neg$(pickup Tree_1 Corner_1 Ti ?tx) v (x accessible ?tx)

The world state is:

(Kato hasfreehands Ti)

(Kato isat Corner_1 Ti)

(Tree_1 isat Corner_1 Ti)

(Tree_1 accessible Ti)

We now set up the theorem prover with these clauses, and eventually produce the following clauses:

$\neg$(pickup Tree_1 Corner_1 Ti ?tx) v (Kato isholding Tree_1 ?tx)

$\neg$(pickup Tree_1 Corner_1 Ti ?tx) v (Kato isat Corner_1 ?tx) v (Kato = Tree_1)

This is our new world description. The (Kato = Tree_1) literal would be stripped off by our fact axioms, when we try to expand the state description.

Unfortunately, because of frame problems, this method is not useable as is. My implementation is a modification of this process. Instead of frame axioms, I have an action demon which describes the effects of the action, and then assumes that everything remains the same unless it fails some simple logical tests. This arrangement is more ad hoc, and certainly carries no guarantees, but any method using axioms would have to guarantee the correctness of the axioms.

Regressing an Action

Again, I present a general method for regressing an action. For specific information on my implementation, see chapter 5.

Regression and progression are very similar procedures. We instantiate our chosen action into the relevant axioms, except that this time, we unify the action's second time term against the free time variable found in the current state. We again turn on the theorem prover and stop when all the propositions contain only references to the new, unbound time variable.

If there are no preconditions possible that would achieve the current state, then no new description will be found, and we will not be able to produce a new world description. This corresponds to choosing an unregressable action.

Again, let us look at an example. I will illustrate the putdown action from the robot world. The current-state is:

¬(Kato isat Corner_2 tz) v ¬(Tree_1 isat Corner_1 tz)

One action that could achieve (Tree_1 isat Corner_1 tz) is the (putdown Tree_1 Corner_1 tx tz) action. The associated axioms are:

¬(Kato isat Corner_1 tx) v ¬(Kato isholding Tree_1 tx) v ¬(putdown Tree_1 Corner_1 tx tz) v (Tree_1 isat Corner_1 tz)

¬(Kato isat Corner_1 tx) v ¬(Kato isholding Tree_1 tx) v ¬(putdown Tree_1 Corner_1 tx tz) v (Kato hasfreehands tz)

After going through the theorem prover, we get:

¬(Kato isat Corner_1 tx) v ¬(Kato isholding Tree_1 tx) v ¬(putdown Tree_1 Corner_1 tx tz)

Notice that we did not need in this case to refer to numerous frame axioms.

Determining Which States can Imply Others

If we can find a state on the progress list which implies a state on the regress list, then the algorithm will terminate. Since we can logically test each new state against its brothers on the other list it is not necessary to put the implication function into the knowledge base. However, only one implication will ever be found, and most of the states will be "obviously" incompatible.

Determining which states are "obviously" incompatible and which states need the full weight of the theorem prover to decide is "obviously" the task of a heuristic function. It is for this reason that I have put this function in the Knowledge Base, although, in the end, this function will eventually call in the theorem prover.

Other Functions Inside the Knowledge Base

Other functions which evaluate the usefulness of world states and actions are required to determine the shape of the search. They obviously are heuristic in nature, and thus belong in the Knowledge Base. For details of one implementation see chapter 5.

5. The Kato-Heron Domain

The only working implementation of a knowledge base for my planner is the world of the U of A Kato-Heron robot. To date, the robot has not performed any plans generated by the planner. The robot will be able to perform these plans after the middle level navigation and sonar scan interpretation systems are developed.

The world consists of a rectangular room logically divided into 5 locations: Corner1, Corner2, Corner3, Middle-of-room, and Doorway. Aside from the robot, the only inhabitants of this world are several "sonar trees", which have a large sonar profile and can be picked up and moved by the robot.

This is an extremely simple world, and the robots capabilities do not amount to more than the ability to shuffle the trees back and forth. The complexity will undoubtedly increase when the vision system is installed.

5.1 Choosing, Progressing, and Regressing Actions

Choosing Actions

When looking for actions that may be suitable for progression I check to see if all preconditions are met. For purposes of the Kato-Heron world, these preconditions are detailed as a list of literals attached to the property list of each action's predicate symbol.

This matching is done by conducting a depth first, left to right search for bindings unifying each of the preconditions of an action to some clause in the current state, in a manner very similar to the PROLOG mechanism. Every successful binding is put on the possible action list.

The possible action list is complete when every binding for each action (pickup, putdown, and goto) has been found. Next, these possible actions are evaluated by the heuristics and then the list is ordered.

Seeking actions to regress is not a mirror image of seeking actions to progress. Since the goal state describes only what must be true, it is unlikely that all of the postconditions (again detailed as a list of literals attached to the action predicate symbol) will unify against the goal state. In this case, we look for one postcondition unifying against some literal in the goal description. This corresponds to the action achieving one of the goal literals. In other words, the action does something.

Some of these actions may not be "do-able"; however, only when we regress the action and discover whether it leads to an inconsistent state, can we determine if an action is "do-able" or "regressable". A simple test is used to determine whether there may be an immediate inconsistency in regressing this action. We simply check to see if any postcondition explicitly conflicts with one of the goal literals, by a successful unification of the negation of a postcondition with one of the goal literals. This corresponds to the consequences of the action explicitly disallowing one of the goals, and thus no set of circumstances could exist which would allow us to achieve all of the goals with this particular action.

Again, we make a list of all of the successful matches of all of the actions. This is our possible regress action list. It is then ordered by our heuristic routines.

Progressing an Action

When the action has been chosen by the appropriate methods and heuristics we progress it through the current state by the following heuristic procedure.

1. Copy each clause from the current state onto the new world state, unless it fails a simple

test, which roughly corresponds to something like the "delete lists" of STRIPS and is my attempt at overcoming the major obstacle of the frame problem. Things are carried over unless they have been specifically targeted to be likely to change. This is different from a pure frame axiom approach where nothing carries over from one state to another unless a specific axiom is present to "bridge the gap". In other words, this step acts as a simple default rule of inference where all of the straightforward axioms (ie things like moving the tree doesn't affect the colour of the wall) are condensed into one object. Any of the more elaborate frame axioms are to be flagged by the test not to carry over, and are be dealt with later in the process.

2. Add certain clauses to the new description, which correspond to the action axioms of an action. These are the direct postconditions of the action.
3. Finally, turn loose a series of functions or "demons" which perform pattern matches between the new and the old states, and add and take away clauses from the new state description. These demons perform the function of the "elaborate" frame axioms, and compute the indirect consequences of an action. As an example, ensure that if the robot moves a box, then everything inside the box must also be moved.

Some of the more elaborate consequences of the action are not be determined at this time, as an attempt is made to transport the minimum required information from one state to another. Later, when the world description is expanded using the theorem prover, in conjunction with the axioms describing the laws of the universe, some of the more esoteric consequences are uncovered. For example, if we carry over the fact that the robot is holding some object, we do not have to carry over the fact that the robot's hands are not free.

Regressing an Action

Regression and progression are somewhat similar procedures. However, as we are dealing with the minimum of what must be true, and because the goals are expressed internally in negated form some changes must be made.

Before any clause in the goal can be regressed into a new state, all of the literals of that clause must be regressed. As the goals are in negated form, each separate clause corresponds to a conjunction of goal literals. All of the different clauses in the goal structure corresponds to an "or" structure (ie., disjunctive goals.)

For this reason, I will consider the regression of each clause separately, using the following procedure:

1. Inspect each literal in the clause. If it does not fail a given logical test, then it is written into a possible clause in the new world state. This step corresponds to the "generalized frame axiom" discussed above.
2. Remove each literal which satisfies the postconditions of the action. Replace them with a list of literals describing the preconditions of the action.
3. On any remaining literals loose a series of pattern matching functions which will replace the literal by a set of literals corresponding to the necessary preconditions which after the action will result in the presence of that literal.

If at any time, we cannot carry back to the necessary preconditions of a particular literal then the entire clause is unregressable. If none of the clauses is regressable then that action is not a regressable action. Furthermore, if any clause generates a tautology, either then, or when expanding the description then the entire state is invalid and is removed from consideration.

Actions

There are three basic actions the robot can perform:

1. (putdown treex locationx timex timey)
2. (pickup treex locationx timex timey)
3. (goto locationx locationy timex timey)

The putdown action causes the robot to place whatever it is holding into the same logical location occupied by the robot.

There are two preconditions for the putdown action. First, the robot must be holding something, and second, the robot must be at the location that whatever it is holding is being placed.

The direct postconditions of the putdown action are: that the robot's hand becomes empty; that whatever it was holding is now at the specified location; and that whatever was just putdown is now accessible to the robot. The secondary postconditions are that anything that was accessible to the robot is now behind the object that was just putdown. Everything else remains the same.

The pickup action may be performed when the robot is not holding anything (hasfreehands); is in the same location as the tree; and if the tree is not behind anything (ie., accessible). These are the preconditions of the pickup action.

After the action is performed, the tree will be held by the robot. Furthermore, anything that was behind the newly picked up object is now accessible.

The robot may perform a goto action at any time. It does have a precondition, in that the robot must be at some location, but since the robot will always be somewhere, this precondition is not much of a hindrance.

The performance of the goto action causes the robot to arrive at the specified location. The robot's middle level navigation routines will ensure that obstacles are avoided. There are no other postconditions of this action, everything else remains unchanged.

There is the possibility of the robot performing a null goto action. In other words, the robot moving to the same location as it currently occupies. I could easily guard against this, but it would make the hierarchical movetree operator more difficult.

There is another minor problem with the preconditions and postconditions as given, namely garbage in, garbage out. I do not demand that things be explicitly "pickupable", or be "places". Clearly, Kato cannot pick himself up (bootstrap procedures notwithstanding), but he also cannot do anything about it. Thus if you give the planner the goal (Kato isholding Kato Tg), you will get the plan (pickup Kato Katos-location Ti Tg). This sort of stuff can be easily dealt with by adding in a series of consistency checks before turning the problem over to the planner.

Facts about the World

The Kato-Heron KB contains a variety of facts and properties. It, of course, has the usual identity facts:

$\neg(\text{Kato} = \text{Tree}_1)$, $\neg(\text{Tree}_1 = \text{Tree}_2)$, etc.

It also has some properties, for example:

$\neg(x \text{ behind } y \text{ tx}) \vee \neg(x \text{ accessible tx})$

$\neg(\text{Kato isholding treex tz}) \vee \neg(\text{Kato hasfreehands tz})$

$\neg(x \text{ isat locationx tx}) \vee \neg(y \text{ isat locationy tx}) \vee (x = y) \vee (\text{locationx} = \text{locationy})$

These allow us to "flesh out" the various state descriptions, and are especially useful when checking whether one state implies another.

Some of the information contained in these facts are also contained in the various demons.

Evaluating the Cost of States

The cost of a node is given by the function

U(n) = K1 * Cost-of-plan(n) + K2 * Distance-remaining(n) + K3 * Actions-tried-cost(n)

The function Cost-of-plan(n) gives the cost of progressing or regressing from a root node (either the initial node or the goal node) to node n. This cost is known exactly by adding up the "cost" of each action needed to arrive at that state from a root node.

Currently, the actual cost of any action is established to be 5 units. As the planning proceeds, this cost builds up, eventually overruling everything unless a the constant (K1) governing the build up of costs is set to zero. This ensures that no plan is overlooked entirely.

The function Distance-remaining(n) is a heuristic guess at how close node n is to the other root node, calculated by a weighted sum of differences between n and the root node.

The distance between the initial state and a state on the regress list is measured by testing each literal in the regress state to see if it is implied by the initial state. If it is implied a score of 0 is given, otherwise a score of 1 is given. All of the scores are added up and divided by the number of literals tried. this gives a distance measure between 1 (no literals matched) and 0 (all literals matched). This measure is then multiplied by the heuristic weight, in this case 30.

The distance between the goal state and state on the progress list is measured similarly.

The number 30 was chosen because the practical planning limit to date has been about a plan of 6 actions. Thus, a state with nothing in common with its corresponding root node is placed at the edge of our search.

Action-tried-cost is a heuristic which reflects the loss of utility of a node when many of its most promising actions have been already tried. My implementation merely adds an amount (6) to the nodes cost each time an action is expanded. The number 6 was chosen because it is just a little bit more than the cost of an action. Thus our heuristics will prefer the sons of a node over the continued expansion of that node. This makes our search a little more adventurous.

The constants K1, K2, K3 reflect how the search is to be conducted. For example I can make a breadth first search if I set K1 to 1 and K2, K3 = 0.

Currently, K1 is set at 1, K2 is set at 30 (the event horizon), and K3 is set at 1.

Ordering Actions

The priority ordering between possible actions has two criteria. The first criterion is what action was last performed. For example, if the last action was a pickup action then the next action should not be a putdown action. The second criterion is a form of means-ends analysis. If we are progressing actions then an action is preferred if the postconditions of that action will achieve some part of the goal. If we are regressing actions then an action is preferred if the preconditions match some of the initial conditions.

The actual calculation of the ordering of the actions is straightforward. A non-preferred action is given a penalty of 1000.

A pickup action has a cost of 10 unless the last action performed was a putdown action, in which case it has a cost of 100. A putdown action has a cost of 20 unless the last action performed was a pickup action, in which case it has a cost of 100. Finally, a goto action has a cost of 30 unless the last action performed was a goto action, in which case it also has a cost of 100. These costs are added to the non-preferred penalty to obtain the overall cost of an action.

The list of actions to be tried is then ordered on this basis. There are no differences in cost between different bindings of the same action (eg. two pickup actions) unless one is a preferred action, and the other is not.

5.2 A Simple Example

A typical problem faced by the robot is to get a certain tree into a given corner.

The goal is (Tree$_1$ isat Corner$_2$ Tg), where Tg is some unspecified time. The algorithm negates this to:

$\neg$(Tree$_1$ isat Corner$_2$ tg)

Note that the goal time has become an unbound variable.

The initial conditions are:

(Kato isat Corner$_1$ Ti)

(Kato hasfreehands Ti)

(Tree$_1$ isat Corner$_1$ Ti)

(Tree$_1$ accessible Ti)

The distance measure between the goal and the initial state is 1 (no literals in common). The utility of both nodes is then 30. The algorithm automatically places goal node in front of the initial node, thus making the goal node our "best-bet".

The only possible last action that would achieve the goal is:

(putdown Tree$_1$ Corner$_1$ t$_i$ tg).

After regressing from the goal state, I get:

$\neg$(Kato isat Corner$_2$ t$_1$) v $\neg$(Kato isholding Tree$_1$ t$_1$)

After applying the various fact axioms, I get:

$\neg$(Kato isat Corner$_2$ t$_1$) v $\neg$(Kato isholding Tree$_1$ t$_1$)

$\neg$(Kato isat Corner$_2$ t$_1$) v (Kato hasfreehands t$_1$)

This state (call it node$_1$) has a distance measure of 1. There are still no literals in common (the hasfreehands literal is not negated). It thus has the same measure of distance remaining as the initial state. This cost is added to the cost of already having performed an action to arrive at a cost of 35.

The goal state, having expanded an action is then penalized, giving it a value of 36. The choicelist now consists of the initial state, node$_1$, and the goal state, in that order. node$_1$ is preferred over the goal state because I feel an adventurous search usually succeeds faster than a breadth first search. Therefore the goal node was penalized quite heavily to favour its offspring.

The initial state now becomes our "best-bet". Clearly, it does not imply either node$_1$ or the goal state. The search for actions finds two possible actions, a goto action and a pickup action. Since the pickup action is considerably harder to arrange unless the last action performed was a putdown action, and because neither action can make part of the goal come true, the pickup action is preferred by our heuristics.

The action (pickup Tree$_1$ Corner$_1$ Ti t$_2$) is progressed from the initial state to:

(Kato isat Corner$_1$ t$_2$)

(Kato isholding Tree$_1$ t$_2$)

$\neg$(Kato hasfreehands t$_2$)

Call this state node$_2$. It also has no literals in common with the goal node, and thus has a distance measure of 1. After evaluation, choice-list now consists of node$_1$, node$_2$, initial, and goal states. Node$_1$ is now "best-bet".

Node$_1$ does not link up with any state on progress list. Our routines find two possible actions: a pickup action, and a goto action. Since the last action performed was a putdown action, the goto action is favoured. This gives us:

(Kato isat x t_3)

(Kato isholding Tree$_1$ t_3)

¬(Kato hasfreehands t_3)

This state (node$_3$) is then evaluated. There are no matches with the goal state thus giving it a distance measure of 1. In this case, this heuristic has not been particularly useful, because there is only one goal literal. Choicelist is now node$_3$, node$_2$, initial, node$_1$ and goal. Best-bet now becomes node$_3$.

Since node$_3$ implies node$_2$, we have discovered a solution to the planning problem. The plan is:

(pickup Tree$_1$ Corner$_1$ Ti T_2)

(goto Corner$_2$ T_2 T_3)

(putdown Tree$_1$ Corner$_2$ T_3 Tg)

Note that when the action statements forming the plan are negated the time variables become skolem constants.

5.3 A variant of the Register Exchange Problem

I will illustrate the power of the planner by doing a variant of the register exchange problem. To do this I will create a higher order variant of Kato's world, which includes only one action: the movetree action. This will greatly expand Kato's ability to shuttle trees, while at the same time reducing the complexity of the search.

The Movetree Action

Movetree is a concatenation of the goto, pickup, goto, and putdown actions. The form of the action is :

(movetree treex locationx locationy tx ty).

The preconditions for the action are:

(Kato hasfreehands tx)

(treex isat locationx tx)

(treex accessible tx)

The postconditions of the action are:

(Kato isat locationy ty)

(Treex isat locationy ty)

(Treex accessible ty)

(Kato hasfreehands ty)

The indirect consequences of these actions are the same as those that would occur if Kato performed the sequence of actions that make up the movetree action.

In this example, I will not discuss either the heuristics, or the various other branches of search in any detail.

The problem

The initial conditions for the problem are straight forward. Tree$_1$ is in corner$_1$. Tree$_2$ is in Corner$_2$. The goal is to reverse the position of the trees.

More elaborately, the initial conditions are:

1. (Kato isat Corner$_1$ Ti)
2. (Kato hasfreehands Ti)

3. $(Tree_1$ isat $Corner_1$ $Ti)$

4. $(Tree_1$ accessible $Ti)$

5. $(Tree_2$ isat $Corner_2$ $Ti)$

6. $(Tree_2$ accessible $Ti)$

The Goal when negated is:

1. $\neg(Tree_1$ isat $Corner_2$ $tg)$ v $\neg(Tree_2$ isat $Corner_1$ $tg)$

The first action taken will be a regress action,

(movetree $Tree_1$?locationy $Corner_2$?ty ?tg).

When the regression is performed we get the state:

$\neg(Tree_1$ isat ?locationy ?ty) v $\neg$(Kato hasfreehands ?ty) v

$\neg(Tree_1$ accessible ?ty) v $(Tree_2$ isat $Corner_1$?ty)

Clearly this state ($node_1$) does not match against the initial state. Let us say that the heuristics predict a progression next, and that the chosen action is:

(movetree $Tree_1$ $Corner_1$?locationx Ti ?tx)

The new world state ($node_2$) is:

(Kato isat ?locationx ?tx)

$(Tree_1$ isat ?locationx ?tx)

$(Tree_1$ accessible ?tx)

(Kato hasfreehands ?tx)

$(Tree_2$ isat $Corner_2$?tx)

$(Tree_2$ accessible ?tx) v (?locationx = $Corner_2$)

Clearly this state does not match against the goal state, or the state arising from the regression action ($node_1$).

Finally let us next look at the regression of $node_1$ from the action (movetree $Tree_2$?locationz $Corner_2$?tz ?ty).

This gives us $node_3$:

¬($Tree_1$ isat ?locationy ?tz) v ¬(Kato hasfreehands ?tz) v ¬($Tree_2$ isat ?locationz ?tz) v ¬($Tree_2$ accessible ?tz) v ¬($Tree_1$ accessible ?tz) v ($Tree_2$ isat $Corner_1$?tz)

This state is implied by the progression state ($node_2$) given the bindings to replace ?locationx by $Corner_3$, ?locationy by $Corner_2$ and ?locationz by $Corner_1$, and the knowledge contained in the KB that ¬($Corner_1$ = $Corner_3$), etc.. This illustrates the requirement to have a set of fact and property axioms.

5.4 Disjunctive Goals

I will present a final, trivial example, illustrating the ability of the planner to handle disjunctive goals.

The initial conditions are absurdly simple:

(Kato isat $Corner_1$ Ti)

The goal is almost as simple:

(Kato isat $Corner_1$ Tg) v (Kato isat $Corner_2$ Tg)

This is negated to:

¬(Kato isat $Corner_1$ tg)

¬(Kato isat $Corner_2$ tg)

When the algorithm starts, it almost immediately determines that the goal state is implied by the initial state, and that no plan is required. This example, however trivial, is beyond the ability of most of the STRIPS-like planners because of their inability to express disjunctive goals.

With this logical approach, no special procedures are required to deal with disjunctive goals.

6. Implementation Issues not Concerning the Knowledge Base

The robot planning system is written in Common Lisp. It currently runs on an IBM PC AT and a Sun 3 workstation. All of the system's development was done on the AT, but because of the AT's limited memory, it has been moved to the Sun.

The program is about 800 lines of lisp code, and consumes about 1 meg of memory when running on a small problem.

Data Structures

The central data structure of the algorithm is the node description. It is implemented as a series of properties attached to the property list of the unique symbol assigned to each world description. The primary property is a list of clauses which describe the current world state. No action literals are part of these clauses. The next property is the instantiated action which generated the current world state. On the parent property is the name of the node which together with the action caused the node to come into existence. On the initial node and the goal nodes the action property and the parent property are blank. Getplan uses these two properties to assemble the plan.

The next property is the heuristic value of the node. This gives the (hopefully) likelihood of determining a solution from this node, and is what determines a nodes location in choice-list. Finally there are an ordered list of actions to be tried and a list of actions already tried. These are used by the heuristics to determine what action should next be used when progressing or regressing from that node.

The Theorem Prover

A major component of the bigress algorithm is the theorem prover. The basic method of operation was taken from John De Haan's (1986) theorem prover. It does not use paramodulation as a rule of inference, because the Kato world is simple enough not to need a substitution rule, and would thus add unnecessarily to the complexity of the system. Any of the needed identity relationships are stored as facts within the KB.

The theorem prover tries to find an empty clause. Input consists of a list of clauses and possibly some predigested clauses (see the next section). For every input clause, the prover renames the variables, lexically orders the clauses, and gives it a name. In other words, the prover puts the clause into a canonical form. Then for every literal in the clause it puts the name of the clause, and the number of the literal onto one of two lists. If the literal is positive, it goes onto a list of positive occurrences of that predicate. Otherwise it is placed on the a list of negative occurrences of the predicate. These lists are found on the property list of the predicate.

In this manner we assemble lists of positive and negative occurrences of each predicate. Clearly, a resolution is possible only when we try to resolve a positive occurrence of a predicate against a negative occurrence of that predicate. For every distinct pair of positive and negative occurrences we create an agenda item which says to try resolving these two literals. The agenda is then sorted by looking at the length of the resulting clause, and placing the shorter ones first.

The theorem prover now takes each agenda item in turn, and tries each resolution. If the resolution fails, we proceed to the next agenda item. If the resolution succeeds, and we have not generated the empty clause, we factor, and turn the resulting clause into canonical form. Then we put the names of the parents, and the resulting binding onto the property list of the new clause Next, we add its literals to the positive, and negative occurrence lists. Finally, we make agenda items for each new possible resolution of some literal of this clause against some previous literal, and re-sort the agenda.

If we ever run out of agenda items, the proof attempt fails. If we ever generate the empty clause, we backtrack along the successful chain of resolutions and return the binding.

Expanding World Descriptions

The matter is a little different when the theorem prover is used to expand the world description. This is very important as a variety of implicit relationships must be brought to the surface if our heuristic progress and regress functions are to work properly.

In this case, we take the current world description and add it to the axioms governing the functioning of the world. As these axioms have already be processed, their individual interactions are not placed onto the agenda. Only the possible resolutions between axioms and world clauses, or between clauses are placed on the agenda.

When there are no more resolutions, or after a certain amount of time, the expansion process stops. All new clauses generated are then added to the world description.

If the expansion process ever generates the empty clause, it terminates with an error message.

7. Conclusions

As discussed earlier, many implemented planning systems have had fundamental logic problems when attempts were made to expand their domain. Logic planning systems, on the other hand, have always foundered on the combinatorial explosion of complicated problems. This thesis has tried to develop a middle approach. Producing a working planner for the Kato-Heron robot, and yet designing the algorithm with enough flexibility to allow for easy extension into more complex domains.

After a brief introduction, this thesis discussed some robots and gave some background to the U of A robot. It became clear that the planning issues confronting the Kato-Heron robot were central to the development of mobile robot system.

The third chapter discussed some planning systems, and expanded on some of the central features of the planning problem.

The fourth chapter detailed the first order logical groundwork which allows the statement of plans. It further discussed the deterministic bigress algorithm, and gave proofs of its correctness, and completeness of a breadth-first version of the algorithm. It also discussed some of the invariant requirements of the algorithm.

The fifth chapter discussed how this planning algorithm applies to the Kato-Heron robot. In particular, it detailed the various heuristic functions used by the system, and gave a few examples of how the system works.

The sixth chapter discussed some of the implementation issues not related to the Knowledge base. It concentrated on discussing the theorem prover.

This last chapter will discuss the strengths and weaknesses of the present planning system, followed by some remarks on how to expand the system. Finally, some concluding remarks will be made.

7.1 Strengths and Weaknesses

The Kato-Heron world is very straightforward. A planner to allow the U of A robot to function in its environment need not have been so elaborate. There are a number of disadvantages with the present system.

Firstly, the planner can currently only plan to a depth of about 5 or 6 actions. More than this amount usually causes a stack overflow error. This explosion of the search space can be made worse if my simplistic heuristic ordering functions choose the wrong ordering.

Secondly, the planner cannot create conditional plans. This makes the utilization of sensory feedback difficult. This is an important extension if the planner is to make the Kato-Heron robot at all useful.

Another problem is that the search procedures will eventually try out ~~every ordering~~ of conjunctive goals, even if there is little or no goal interaction. For example, if we want to place $Tree_1$ in $Corner_1$, and $Tree_2$ in $Corner_2$, etc. the algorithm will consider the emplacement of any one of the trees as a possible last action. The emplacement of any remaining tree is then a further possible action. This can give rise to a very large search tree.

One possible way to avoid these problems is via an adventurous search. As long as we are in some sense "succeeding" we try to maintain a depth first search. It is only when things look equally bad that we resort to a breadth first search. This of course, relies heavily on the cleverness of our heuristics.

In-fact, a straightforward programming *tour de force* would probably have produced a somewhat flashier, if more *ad hoc* (and hence disposable) planner.

I believe, however, that the advantages greatly outweigh the disadvantages. The greatest contribution of the planner is that it is possible, if somewhat unwieldy, to create a useable planner based on first order predicate calculus. The power of modal logic is not required.

Furthermore, the planner avoids the full force of the frame problem by the use of a default rule of inference, while at the same time it performs much better than other heuristic systems when considering disjunctive goals, or goals with heavy interaction such as the variant of the register exchange problem.

7.2 Expanding Bigress

Another advantage of the present system lies in its expandability. This section will explore some obvious extensions to the algorithm, and will conclude with some possible applications.

Parallel Bigress

The Bigress algorithm is almost a natural for parallel implementation. This is important as any serious problem solving will probably be beyond the reach of most serial machines.

The algorithm essentially builds a double tree shaped search graph. Thus searching along different branches of these trees can be done almost entirely in parallel. Progressing, and regressing actions are quite independent, but the choice-list structure would have to be overhauled to take advantage of the parallelism.

In addition, as intercommunication between trees is only required when determining whether one state can imply another, the implications could be performed in parallel as long as we had arranged a semaphore on the progress and regress-list data structures to tell us what states have been checked.

Hierarchical Planning

With its logic based approach to planning, this algorithm is particularly vulnerable to trying to search too large a search space, and hence failing miserably. A necessary improvement would be to reduce the search space, and as Kautz says the greatest efficiencies in search can be gained in the use of higher order planning.

As long as it is possible to accurately describe the effects of a higher order action, it does not matter to bigress whether the action axioms are "primitive" in some sense. We can just use bigress to derive a high order plan. This plan would be a series of high order actions. Then, for every action in this series, we would call bigress to form a lower order plan using the results of the higher order actions as goals for the lower order planning. Of course these new calls to bigress would be with a lower level knowledge base detailing the effects of the more primitive action. These lower order plans would be strung together to form a new plan. This new plan might even be used as a basis for further, even lower order, planning.

In some instances, these lower level actions can easily be preplanned. This is clearly the case with the Movetree action described earlier. There we had a strict concatenation of 4 primitive actions. In other cases, a higher level action would have to be planned out in lower level detail every time the higher level action is postulated. A simple example of this is the "cook dinner" operator. So many different things can be done to make so many different dinners that it would be foolish to try to precompute the low level actions needed to cook a meal. All the same, we can make some pretty clear logical statements about the higher level 'cook dinner' operator.

Conditional Planning

The current planner implicitly assumes that all preconditions required for progressing an action will be known to be true or not before execution begins. This is clearly not always the case.

This means that we must change our requirements of what actions can be progressed. You will recall that we had insisted on all preconditions being true before allowing that an action be progressable. Now we can assume that one or more preconditions will be true. Of course we must ensure that our assumption still maintains a consistent world state. We can now carry on and derive a plan that will work if all of our assumptions turn out to be true.

If we feel that the assumptions are as likely not to be true then we can assume that the opposite is true, and derive a plan with the opposite assumption. This other plan may involve taking steps to achieve the validity of our assumption. Since one or the other must be true, we can now join these plans with a test performed at execution time. If the test shows the assumption correct we chose one branch of the plan, otherwise we perform the other branch.

This testing procedure is a perfect place to place sensory input. For example, we may know that a tree is either in Corner1 or in Corner2. Testing Corner1 to determine the presence or absence of the tree will tell us what actions to perform.

Of course, if our heuristics told us that a precondition was likely to be true, we could just develop the likely branch. We would just assume that the precondition was true. The other branch would be merely expressed as a call to our planning algorithm, to be executed if the execution of our plan failed.

This ability to assume is a very powerful feature. If something isn't immediately available, just assume it. This could lead to a swarm of conditional plans. A more reasonable approach would probably be place restrictions on what could be safely assumed. One possible option would allow us to only assume certain preconditions, such as the locations of trees, and perhaps certain internal states of the robot. Other preconditions, such as whether certain doors are locked are not would have to be known before the planner started.

Iterative Planning

With the concepts of hierarchical planning and conditional planning, it is possible to produce iterative planning. Simply create a hierarchical plan be composed of a conditional plan in which part of the condition contains a reference to the hierarchical plan itself. Of course, certain precautions against infinite recursion must be taken.

The exact mechanisms and requirements for this sort of planning are not well explored.

Concurrent Planning

As mentioned earlier, this first order format for expressing plans with time arguments is almost a natural for planning concurrent actions. We would have to extend the semantics of the language, as well as provide some special purpose time inference mechanisms, but the basic structure of the algorithm would remain the same.

Planning Planning

We typically think of hierarchical planning being composed of higher order actions. In order to save time and effort, we usually try to precompute the lower order actions as they tend to fall into distinct patterns.

Another approach would be to regard hierarchical planning as a primitive form of planning the planning. Instead of stacking blocks, and moving trees, we could visualize it as moving preconditions and considering interactions. This view may be able to reduce the search space even more.

Quite often, humans can identify the hurdles of the planning process quite easily. We then devise heuristics for the planner that guide our search accordingly. After all, given a set of goals we can usually say with some precision what certain low level actions must be performed. The order may not be known, and other actions may be interspersed, but certain actions will be performed.

In conjunction with planning speech acts, James Allen (1987), has written on 'metaplanning' and has proposed some specific meta-level actions for manipulating and modifying plans.

Clearly, planning planning is an important field needing much more research.

Future Applications

The planner as it stands is unsuitable for immediate application to real-world problems. It would have to be a part of a higher level system which would invoke the planner when required. Other parts of this system would include sensory perception abilities. As such, the planner could easily be used in semi-autonomous robots such as spacecraft, deep sea robots, or other hostile environment robots where human supervision is intermittent.

Other applications include robots which have highly variable tasks within a regular world. An example of this is a courier robot inside of an office block. A final application is a self-directed robot which chooses its own goals given the situation it is currently facing, and then must have some method of achieving those goals.

7.3 Concluding Remarks

The planning system as implemented creates only simple plans. Issues such as the frame problem are dealt with in a heuristic manner, while maintaining the flexibility of the logical approach. Because of this flexibility, the system is able to outperform some of the STRIPS-like systems on a number of very simple problems.

The major achievement of this planner is to demonstrate that a planner based on first order logic can be feasible, while maintaining its generality and expandability. This allows the system to be incrementally, and elegantly expanded to cover larger and larger problems as time, and computing resources allow.

Bibliography

Aleksander, I.; Burnett, P. 1983. *Reinventing Man* Kogan Page.

Allen, J.; Litman, D. 1987 "Plans, Goals, and Natural Language" *Special Issue on Natural Language Processing of the Proceedings of the IEEE* Forthcoming.

Brauzil, B.; Briot, M.; Ribes, P. 1983 "A Navigation Sub-System Using Ultrasonic Sensors for the Mobile Robot Hilare." *Proceedings of the 1st International Conference on Robot Vision and Sensory Controls.* Stratford-upon-Avon, U.K.

Chatila, Raja. 1982 "Path Planning and Environment Learning in a Mobile Robot System." *European Conference on Artificial Intelligence.* Orsay, France

Coles, L.; Robb, A.; Sinclair, P.; Smith, M.; Sobek, R. 1975 "Decision Analysis for an Experimental Robot with Unreliable Sensors." *Proceedings of the Fourth International Joint Conference on Artificial Intelligence.* MIT, Cambridge, MA.

De Haan, John. 1986. *Theorem Proving in a Topically Ordered Semantic Net* Masters Thesis, University of Alberta, Edmonton.

Doyle, J. 1979. "A Truth Maintenance System." *Artificial Intelligence* 12(3)

Elfes, A. 1987. "Sonar-Based Real-World Mapping and Navigation" *IEEE Journal of Robotics and Automation* Vol. RA-3, No 3.

Erman, L.D., Hayes-Roth F., Lesser V.R., Eddy E.R. 1980. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty" *Computing Surveys,* Vol. 12, No. 2

Fikes, R. E.; Nilsson, N. 1971. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving" *Artificial Intelligence* (3/4), pp 189-206.

Giralt, G.; Sobek, R.; Chatila, R. 1979. "A Multi-Level Planning and Navigation System for a Mobile Robot; A first Approach to HILARE" *Proceedings of the Sixth International Joint Conference on Artificial Intelligence.* Tokyo, Japan.

Green, C. 1969 "Application of Theorem Proving to Problem Solving" *International Joint Conference on Artificial Intelligence* Washington, D. C. pp 219-239

Gruver, W. A.; Soroka, B. I.; Craig, J. J.; Turner, T. L. 1983 "Evaluation of Commercially Available Robot Programming Languages" *13th International Symposium on Industrial Robots* Chicago.

Inoue, Hirochika. 1985. "Building a Bridge Between AI and Robotics" *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* Los Angeles, California.

Kanade, T.; Thorpe, C. 1985. "CMU Strategic Computing Vision Project Report: 1984 to 1985," Carnegie-Mellon University, Pittsburgh, PA. CMU-RI-TR-86-2, Nov.

Kautz, Henry. 1982 "Planning Within First-Order Dynamic Logic" *Proceedings of the CSCSI Conference* U of Saskatchewan,

Lewis, R.; Bejczy, A. 1973 "Planning Considerations for a Roving Robot with Arm" *Third International Joint Conference on Artificial Intelligence* Stanford, CA.

Lewi[illegible] 975 *Cy[illegible]f Unifiability and Decidability by Resolution*, Tech Report, Aiken C[illegible]ation [illegible]boratory, Harvard University, Cambridge, MA.

Litvin[illegible]t, V. 1977. "A Proof Checker for Dynamic Logic." *Proceedings of the Fi[illegible]national Joint Conference on Artificial Intelligence* MIT, Cambridge, MA.

McCorduck, P. 1979. *Machines Who Think* San Francisco. W.H. Freeman and Company.

McDermott, Drew. 1976 "Planning and Acting" *Cognitive Science* 2.

Moravec, H. 1984 "Locomotion, Vision, and Intelligence" *Robotics Research: The First International Symposium.* Edited by M. Brady, and R. Paul. MIT Press, Cambridge, MA.

Newell, A.; Shaw, J.; Simon, H. 1960. "Report on a General Problem-Solving Program for a Computer." *Information Processing: Proceedings of the International Conference On Information Processing,* UNESCO, Paris

Nilsson, Nils. 1971 *Problem Solving Methods in Artificial Intelligence* New York: McGraw-Hill.

Nilsson, Nils. 1980 *Principles of Artificial Intelligence* SRI International.

Nitzan, D. 1979. "Flexible Automation Program at SRI." *Proceedings 1979 Joint Automatic Control Conference.* New York: IEEE.

Podnar, G.; Blackwell, M.; Dowling, K. 1984. *A Functional Vehicle for Autonomous Mobile Robot Research,* CMU Robotics Inst., Apr.

Podnar, G. 1986. "The Uranus Mobile Robot," *Autonomous Mobile Robots: Annual Report 1985,* Mobile Robot Lab., Pittsburgh, PA., Tech. Rep. CMU-RI-TR-86-4, Feb.

Raphael, Bertram. 1970 "The Relevance of Robot Research to Artificial Intelligence." *Theoretical Approaches to Non-Numerical Problem Solving,* Banerji, R. and Mesarovic, M. Ed. Berlin and New York: Springer-Verlag.

Raphael, Bertram. 1976. *The Thinking Computer: Mind Inside Matter*. San Francisco: W.H. Freeman and Company

Rich, E. 1983. *Artificial Intelligence* McGraw-Hill.

Rosenschein, Stanley. 1981 "Plan Synthesis: a Logical Approach." *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, University [illegible] British Columbia, Vancouver, B.C.

Sacerdoti, E. D. 1974. "Planning in a Hierarchy of Abstraction Spaces" *Artificial Intelligence*, 5(2), pp 115-135.

Sacerdoti, E. D. 1977. *A Structure for Plans and Behavior* New York: Elsevier.

Smith, M.; Coles, L. 1973 "Design of a Low Cost, General Purpose Robot." *Third International Joint Conference on Artificial Intelligence*. Stanford, CA.

[illegible]oka, B. I. 1983. "What Can't Robot Languages Do?" *13th International Symposium on Industrial Robots* Chicago.

Stacey, R. S. 1986. *Algorithmic Generation of Synthetic Speech* Masters Thesis, University of Alberta, Edmonton.

Stefik, M. 1981. "Planning with Constraints, MOLGEN part 1" *Artificial Intelligence*. 16(2)

Stefik, M. 1981. "[illegible]ing with Constraints, MOLGEN part 2" *Artificial Intelligence*. 16(2)

Sussman, G. J. 1975. *A Computer Model of Skill Acquisition*, MIT Press, Cambridge, Mass.

Tate, Allen. 1977. "Generating Project Networks." *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* MIT, Cambridge, MA.

Vere, Steven. 1983. "Planning In Time: Windows and Durations for Activities and Goals." *IEEE Transactions on Pattern Analysis and Machine Intellige[illegible]* [illegible](3).

Thompson, Alan. 1977 "The Navigation System of the JPL Robot" *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* MIT, Cambridge

Waldinger, R.J. 1977. "Achieving Several Goals Simultaneously." *Machine Intelligence 8* Ellis Horwood, Chichester.