

Not In System
4-6-73

DRA

Final Report

Covering the Period 7 October 1969 to 7 October 1970

RESEARCH AND APPLICATIONS-- ARTIFICIAL INTELLIGENCE

By: BERTRAM RAPHAEL

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
600 INDEPENDENCE AVENUE, S.W.
WASHINGTON, D.C. 20546

Attention: MR. SAMUEL A. ROSENFELD/RET

CONTRACT NAS12-2221

Sponsored by

ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C. 20301
ARPA ORDER 1058 AMENDMENT 1



STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

(NASA-CR-131785) RESEARCH AND
APPLICATIONS: ARTIFICIAL INTELLIGENCE
Final Report, 7 Oct. 1969 - 7 Oct. 1970
(Stanford Research Inst.) 175 p

N73-72140

Unclas
00/99 17653



STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Final Report

November 1970

Covering the Period 7 October 1969 to 7 October 1970

RESEARCH AND APPLICATIONS-- ARTIFICIAL INTELLIGENCE

By: BERTRAM RAPHAEL

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
600 INDEPENDENCE AVENUE, S.W.
WASHINGTON, D.C. 20546

Attention: MR. SAMUEL A. ROSENFELD/RET

CONTRACT NAS12-2221

SRI Project 8259

Approved by:

DAVID R. BROWN, *Director*
Information Science Laboratory

BONNAR COX, *Executive Director*
Information Science and Engineering Division

Sponsored by

ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C. 20301
ARPA ORDER 1058 AMENDMENT 1

Copy No. 65.....

ABSTRACT

This is the final report for the most recent year of a continuing program of research in the field of artificial intelligence. This work follows previous projects that resulted in the design, construction, and demonstration of a "first generation" robot system. The work reported here consists of new research aimed at the development of a more sophisticated "second generation" robot. Although the robot vehicle itself will be essentially unchanged, it will be controlled by a completely new computer hardware and software system. In particular, this report contains detailed descriptions of the computer configuration and the bottom-level software design, two new bases for problem-solving systems (called STRIPS and QA4), and new directions in visual scene-analysis techniques.

CONTENTS

ABSTRACT	iii
LIST OF ILLUSTRATIONS	vii
LIST OF TABLES	ix
I INTRODUCTION	1
II BACKGROUND	1
III REPORT OUTLINE	2
IV SUMMARY OF RESULTS	4
A. Computer System	4
B. Bottom-Level Software	4
C. Problem-Solving Research	5
D. Vision	6
V PUBLICATIONS AND PRESENTATIONS	7
A. Publications	7
B. Presentations	10
APPENDIX A--COMPUTER CONFIGURATION	13
APPENDIX B--BOTTOM-LEVEL PDP-10 SOFTWARE FOR THE SRI ROBOT	19
APPENDIX C--THE FRAME PROBLEM IN PROBLEM-SOLVING SYSTEMS	45
APPENDIX D--STRIPS: A NEW APPROACH TO THE APPLICATION OF THEOREM PROVING TO PROBLEM SOLVING	67
APPENDIX E--QA4 WORKING PAPER	107
APPENDIX F--SOME CURRENT TECHNIQUES FOR SCENE ANALYSIS	141

ILLUSTRATIONS

APPENDIX A

Figure 1	PDP-10 Configuration	17
----------	--------------------------------	----

APPENDIX D

Figure 1	Flowchart For the Strips Executive	88
Figure 2	Configuration of Objects and Robot For Example Problem	90
Figure 3	Search Tree For Example Problem	97

APPENDIX E

Figure 1	Iterative Fibonacci Program	131
----------	---------------------------------------	-----

APPENDIX F

Figure 1	Three Corridor Scenes	148
Figure 2	Results of Merging Heuristics	149
Figure 3	A Simple Scene	157
Figure 4	A More Complicated Scene	161
Figure 5	The Analysis Tree	163
Figure 6	Basic Flowchart For Landmark Program	167
Figure 7	Landmarks	168

TABLES

Table I Major System Changes 3

Appendix F

Table I Correspondence Between Boundary Setment Configurations and Characters Used in Printout 150

Table II Regions That Are Legal Neighbors 153

Table III Hypothetical Region Scores 162

I INTRODUCTION

This is the final report for the most recent year of a program of research in the field of artificial intelligence. The present project began in October 1969 as a direct continuation of work performed under a previous contract.* It is expected that the work will be continued under new support. Therefore this is a report on the present status of a continuing research program.

An Interim Scientific Report[†] was prepared in April 1970 which describes activities during the first six months of this project. This present report therefore emphasizes more recent work and is designed to augment, rather than replace, the Interim Report.

II BACKGROUND

In our previous projects, research was focused on the application of techniques of artificial intelligence to the control of a mobile automaton in a realistic laboratory environment. This work culminated late in 1969 in some demonstrations of a complete automaton system, which is documented in numerous papers and reports and in a 25-minute motion picture entitled "Shakey: A First Generation Robot." As a result of that work we discovered a variety of limitations, both in the capabilities of our computer facility and, more important, in the techniques we were using for various conceptual portions of the system design.

* Contract F30602-69-C-0056 with the Advanced Research Projects Agency and the Rome Air Development Center.

† L. J. Chaitin et al., "Research and Applications--Artificial Intelligence," Interim Scientific Report, Contract NAS 12-2221, Stanford Research Institute, Menlo Park, California (April 1970).

During the past year we have been converting our hardware to a new, more powerful computer configuration. As a result, the physical robot vehicle has not been available for experimentation. This change-over period provided us with an excellent opportunity to make basic studies in several areas, and thereby redesign the major software components of the system. As a result of this redesign, we have now established the framework for our second-generation robot system, which will be implemented during the coming year. Table I summarizes the major departures between our first and second robot generations. The remainder of this report describes the technical considerations that resulted in various aspects of the second-generation framework.

III REPORT OUTLINE

Instead of preparing the customary single, large, integrated report, we have decided to make this final report consist of a brief document that summarizes our accomplishments, supported by several appendices, including some Technical Notes, that contain the technical details of the work. As discussed in Section II, the bulk of our work this year has consisted of several separate basic studies. The appended Technical Notes, also available separately, provide documentation for the present status of each of these studies. In the year ahead we plan to integrate some of these results.

Section IV contains summaries of our progress during the past year and provides a guide to the contents of the appendices. Finally, Section V contains descriptions of publications and presentations by our staff members during this past year with the support of this project.

Table I

MAJOR SYSTEM CHANGES

<u>Subject</u>	<u>First Generation</u>	<u>Second Generation</u>
Computer Facility	SDS-940 Core memory 64K, 24-bit words, 0.5M word drum	PDP-10 Core memory 192K, 36-bit words, 1.5M word drum, 20M word disc, PDP-15 periph- eral computer
Problem Solving	QA3 Theorem-proving approach, using state variables	STRIPS A problem-solving executive that uses a theorem prover as a subroutine
Vision	Analysis based on local detection of line segments, followed by global considerations	Analysis based on regions and early use of global infor- mation
Implementation	Dual grid and list models; parallel use of FORTRAN and LISP systems	Single "tuple" model; LISP operating system in control; new QA4 language under devel- opment

IV SUMMARY OF RESULTS

A. Computer System

A new computer facility, purchased by the government (partly under Contract F30602-69-C-0056 with Rome Air Development Center) for the use of this project and its possible successors, was procured and installed. This installation is now essentially complete. We expect to have the complete system operational within a few months. Meanwhile, the robot and camera hardware have been attached to the new system and are almost completely operational. Appendix A describes the new configuration.

B. Bottom-Level Software

Transferring the SRI robot from the SDS-940 to the PDP-10/PDP-15 system has given us the opportunity to replace the rather complicated robot software interface on the 940 with one more tailored to the user's view of what the robot does. By carefully defining the actions that the computer can ask the robot to perform, their possible consequences, and so on, we hope to achieve logical clarity and, at the same time, enhanced usefulness. Appendix B describes the "front-side" interface to the bottom-level robot software (i.e., the interface through which the routines are called as operators or subroutines by other software in the robot program hierarchy). It also describes, when appropriate, the "back-side" interface between the bottom-level software and the PDP-15/robot complex, and the internal structure of the bottom-level software itself.

C. Problem-Solving Research

Problem solving was done in the first-generation robot system by QA3, a theorem-proving and question-answering system based upon first-order predicate calculus. Although mathematically elegant, the approach used to generate plans of action became extremely inefficient in all but the most trivial situations. The major difficulty was due to the "frame problem," the problem of creating and maintaining an appropriate informational context or "frame of reference," which is discussed in detail in Appendix C. Additional difficulties arise because QA3 fails to distinguish between two phases of problem-solving activity: planning courses of action, and executing the resulting plans. (The particular importance of the execution phase of robot problem-solving activity was discussed by J. H. Munson in Appendix F to the Interim Report.)

A new problem-solving system called STRIPS has now been defined and is described in Appendix D. It consists of an executive that makes use of a theorem-proving program, as a subroutine, to make various tests such as whether a proposed action is applicable to a given situation. STRIPS contains, however, considerably more flexibility than would a theorem prover alone.

The QA3.5 system, an upgraded version of QA3, can be modified to interface with STRIPS. (QA3 is a question-answering system, described in the cited final report of the ARPA-RADC project, that contains a theorem-proving program for first-order predicate calculus.) In addition, this project has been partially supporting the development of a new system called QA4. Originally conceived as a next-generation theorem

prover based upon higher-order logic, QA4 has evolved into the specification of a programming language particularly well-suited for use in the design of new theorem-proving and problem-solving systems. Future versions of STRIPS and QA3, as well as separately supported projects for automatic program construction or verification, will probably be programmed in the QA4 language. The status of the QA4 development is described in Appendix E.

D. Vision

The vision part of the first-generation system was based on the detection of line segments in the picture. Scene analysis was accomplished by a decision-tree program whose structure reflects the known constraints and relations in the robot's visual world. This program added information to the robot's model but did not use any information previously stored in that model.

General scene analysis in the second-generation system will be based on the detection of regions in the picture. A quite complete library of routines for region analysis has been coded. Current plans for scene analysis envision replacing the decision tree by a search procedure that will allow an explicit, formal, and easily modifiable description of the known constraints and relations. A detailed description of this approach is given in Appendix F.

As the robot's model of the world becomes more complete, the role of vision changes from one of exploring an unknown world to one of providing visual feedback. A typical task here is that of sighting landmarks and using them to update the robot's knowledge of its own location and orientation. A program that accomplishes this is

described in Appendix F. The use of specific information in the model to aid scene analysis is expected to play a prominent role in the second-generation system.

V PUBLICATIONS AND PRESENTATIONS

A. Publications

Following is a list of technical notes and papers generated by the staff of the Artificial Intelligence Group of Stanford Research Institute with the support of this project:

- (1) J. Munson, "A LISP-FORTRAN-MACRO Interface for the PDP-10 Computer," Technical Note 16 (November 1969).
- (2) C. Brice, C. Fennema, and S. Weyl, "AROS--Algorithms for Partitioning a Picture," Technical Note 18 (January 1970).
- (3) J. Munson, "The SRI Intelligent Automaton Program," Technical Note 19 (January 1970); published in Proc. First Natl. Symp. Industrial Robots, pp. 113-117 (1970).
- (4) R. Duda and P. Hart, "Experiments in Scene Analysis," Technical Note 20 (January 1970); published in Proc. First Natl. Symp. Industrial Robots, pp. 119-130 (1970).
- (5) L. Coles, "Bibliography of Literature in the Field of Robots," Technical Note 23 (March 1970).
- (6) R. Yates and B. Raphael, "Resolution Graphs," Technical Note 24 (March 1970); to be published in Artificial Intelligence, December 1970.

- (7) J. Ellis and L. Chaitin, "PDP-15 Simulator," Technical Note 25 (April 1970).
- (8) R. Waldinger, "Robot and State Variable," Technical Note 26 (April 1970).
- (9) R. Kling, "Some Remarks on Resolution Strategies," Technical Note 28 (April 1970).
- (10) J. Munson, "A Cost-Effectiveness Basis for Robot Problem Solving and Execution," Technical Note 29 (January 1970).
- (11) B. Raphael, "Robot Problem Solving without State Variables," Technical Note 30 (May 1970).
- (12) J. Munson, "The SRI Robot as a Candidate Domain for Vocal Conversation with a Computer," Technical Note 31 (May 1970).
- (13) D. Luckham (Stanford University) and N. Nilsson (SRI), "Extracting Information from Resolution Proof Trees," Technical Note 32 (June 1970); to be published in Artificial Intelligence.
- (14) B. Raphael, "The Frame Problem in Problem-Solving Systems," Technical Note 33 (June 1970); published in Proc. Adv. Study Institute.
- (15) Z. Manna (Stanford University) and R. Waldinger (SRI), "Towards Automatic Program Synthesis," Technical Note 34 (July 1970).
- (16) J. Munson, "Bottom-Level PDP-10 Software for the SRI Robot," Technical Note 35 (August 1970).

- (17) R. Duda and P. Hart, "A Generalized Hough Transformation for Detecting Lines in Pictures," Technical Note 36 (August 1970); to be submitted for publication in the IEEE Trans. Computers.
- (18) R. Kling, "SRI--TRACE Package for PDP-10 LISP," Technical Note 37 (September 1970).
- (19) C. Rosen, "An Experimental Mobile Automaton," Technical Note 39 (July 1970); to be published in Proc. 18th Conf. Remote Systems Technology.
- (20) L. Coles, "An Experiment in Robot Tool Using," Technical Note 41 (October 1970); paper presented at IEEE Systems Science and Cybernetics Conference (abstract published in Proceedings).
- (21) J. Rulifson, J. Derksen, and R. Waldinger, "QA4 Working Paper," Technical Note 42 (October 1970).
- (22) N. Nilsson and R. Fikes, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Technical Note 43 (October 1970).
- (23) R. Kling, "Design Implications of Theorem-Proving Strategies," Technical Note 44 (October 1970).
- (24) C. Brice and J. Derksen, "The QA3 Implementation of E-Resolution," Technical Note 45 (October 1970).
- (25) R. Duda, "Some Current Techniques for Scene Analysis," Technical Note 46 (October 1970).

B. Presentations

Following is a list of presentations made by staff members of the Artificial Intelligence Group during the period of this contract:

- (1) L. Coles, "An Overview of the SRI Robot Project," Workshop and Symposium on Robotics, North American Rockwell Corporation, Thousand Oaks, California, October 9, 1969; talk and movie.
- (2) R. Yates, "Techniques for Robot Problem Solving," Workshop and Symposium on Robotics, North American Rockwell Corporation, Thousand Oaks, California, October 9, 1969; talk.
- (3) L. Coles, "The SRI Robot Project: An Overview," ASME meeting, November 25, 1969; talk and movie.
- (4) B. Raphael, "The Robot Project at Stanford Research Institute," University of Illinois, Department of Computer Science, Urbana, Illinois, January 5, 1970; talk and movie.
- (5) B. Raphael, "Recent Results in Automatic Theorem Proving," University of Illinois, Department of Computer Science, Urbana, Illinois, January 6, 1970.
- (6) R. Duda, "Vision Programs for a Robot," IEEE Systems Science and Cybernetics Group meeting, January 22, 1970; talk and vision movie.
- (7) N. Nilsson, "Robot Research at the Stanford Research Institute," Oregon State University, Department of

- Electrical and Electronics Engineering, Corvallis, Oregon, March 20, 1970; talk and movie.
- (8) B. Raphael, "The Robot Project at Stanford Research Institute," University of Alberta, Department of Computing Science, Edmonton, Canada, April 1, 1970; talk and movie.
- (9) B. Raphael, "The Robot Project at Stanford Research Institute," IPSOC meeting, Vancouver, Canada, April 2, 1970; talk and movie.
- (10) B. Raphael, "The Robot Project at Stanford Research Institute," University of Saskatchewan, Department of Computational Science, Saskatoon, Canada, April 2, 1970; talk and movie.
- (11) B. Raphael, "The Robot Project at Stanford Research Institute," University of British Columbia, Department of Computer Science, Vancouver, Canada, April 3, 1970; talk and movie.
- (12) C. Rosen, Mullard Research Laboratories, Redhill, England, June 24, 1970; movie.
- (13) C. Rosen, University of Edinburgh, Department of Machine Intelligence and Perception, Edinburgh, Scotland, June 26, 1970; movie.
- (14) B. Raphael, "Robot Research," Seminar on Artificial Intelligence, Dubrovnik, Yugoslavia, June 29-July 11, 1970; talk and movie.

- (15) L. Coles, "Natural-Language Processing," and "Intelligent Robots," Seminar on Artificial Intelligence, Dubrovnik, Yugoslavia, June 29-July 11, 1970; talk and movie.
- (16) B. Raphael, "The Frame Problem in Problem-Solving Systems," Advanced Study Institute on Artificial Intelligence and Heuristic Programming, Lake Como, Italy, August 2-14, 1970; talk and movie.
- (17) B. Raphael, CERN, Geneva, Switzerland, August 31, 1970; talk and movie.
- (18) R. Duda, National Conference, Association for Computing Machinery, New York, New York, September 1-3, 1970; movie.

APPENDIX A

COMPUTER CONFIGURATION

by

Leonard J. Chaitin

The Artificial Intelligence Group computer complex consists of the following parts:

- PDP-10 computer and peripherals
- PDP-15 computer and peripherals (including the robot)
- An interprocessor buffer to connect the two computers.

These are interconnected as shown in Figure 1.

The PDP-10 system has 192K (K=1024) words of 36-bit memory. 32K is DEC MD10 memory. The rest is Ampex RG10 memory, consisting of one 32K memory with interface and one 128K memory interface and four modules of 32K each. All memory has four ports. These are occupied by:

- PDP-10 central processor
- DF10 data channel
- Bryant drum controller
- DA25C interface.

The Bryant drum is a high-speed autolift drum which has a 1.5-million-word capacity. It is planned that it will be used for swapping and some system files. The drum controller interfaces directly into the memory rather than going through a data channel.

The DF10 data channel is used to handle I/O from two peripherals: the disk pack drives and the TV A/D converter.

The interface between the disk pack drives and the DF10 data channel was built by Interactive Data Systems, Inc.

The disk pack drives are manufactured by Century Data Systems and handle the 20-surface disk packs. This means that each disk pack has a 5-million-word capacity. The packs themselves are manufactured by Caelus Inc. The disk pack system is used as secondary storage.

Currently, we are also using one disk pack drive as a swapping device for the time-sharing system.

The TV A/D converter is an SRI-designed and -built device. It handles data from the robot TV camera at a rate of one word every 1.5 microseconds. It is capable of processing either 120x120 or 240x240 pictures with 32 levels of gray scale.

The DA25C is the PDP-10 side of the interprocessor buffer. It handles data at one 36-bit word every 8 microseconds. We have programmed it such that the PDP-10 is always in control and can interrupt any transmission in order to initiate one of its own.

The DA25D is the PDP-15 side of the interprocessor buffer. Each PDP-10 word is split into two PDP-15 words (18 bits each). It also does the reverse operation. It operates on the PDP-15 I/O bus as a single-cycle device; however, its internal logic uses three cycles per word.

The PDP-15 has 12K of core memory and an I/O processor. All devices are "daisy chained" on the I/O bus. These include an Adage display, paper tape, DEC tape, A/D converter, D/A converter (not yet delivered), ARPA network IMP (not yet implemented), and the SRI robot.

The Adage display provides a high-speed graphics capability. It will be refreshed from the PDP-15 core. The display lists will be prepared in the PDP-10 and executed from the PDP-15. Capabilities include incremental mode, print mode, dotted lines, and intensity control.

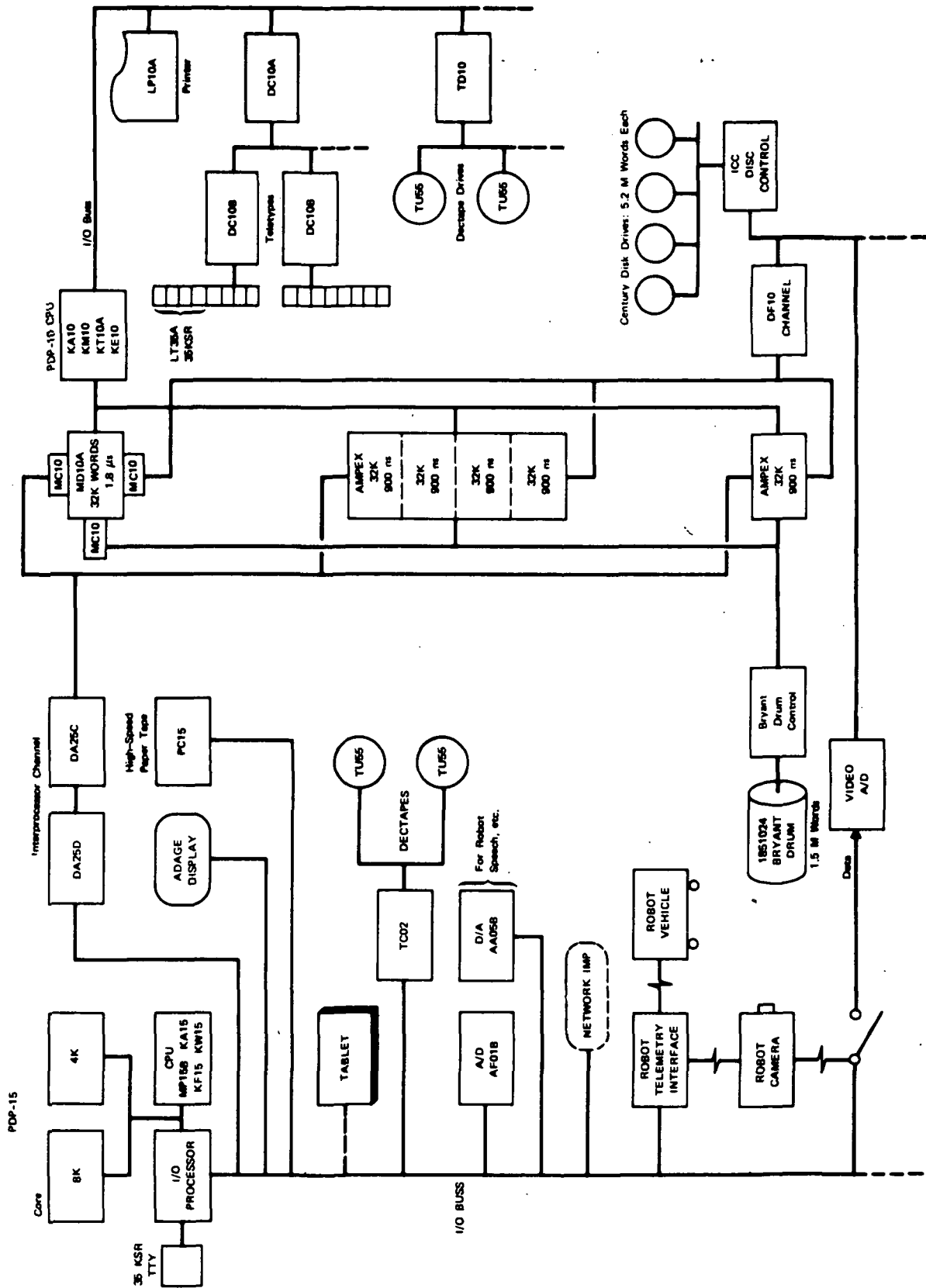


FIGURE 1 PDP-10 CONFIGURATION

APPENDIX B

BOTTOM-LEVEL PDP-10 SOFTWARE FOR THE SRI ROBOT

August 1970

BOTTOM-LEVEL PDP-10 SOFTWARE FOR THE SRI ROBOT

by

John H. Munson

Artificial Intelligence Group

Technical Note 35

SRI Project 8259

This research is sponsored by the Advanced Research
Projects Agency and the National Aeronautics and
Space Administration under Contract NAS 12-2221.

Transferring the SRI robot from the SDS-940 to the PDP-10/PDP-15 system has given us the opportunity to replace the rather obscure and complicated robot software interface on the 940 with one more tailored to the user's view of what the robot does. By carefully defining the actions the computer can ask the robot to perform, their possible consequences, and so on, we hope to achieve logical clarity and, at the same time, enhanced usefulness. This note describes the "front-side" interface to the bottom-level robot software (i.e., the interface through which the routines are called as operators or subroutines by other software in the robot program hierarchy). It also describes as appropriate, the "back-side" interface between the bottom-level software and the PDP-15/robot complex, and the internal structure of the bottom-level software itself.

THE ROBOT ACTIVITIES

The robot presently has four degrees of mechanical freedom: it can tilt and pan its "head," containing the TV camera and rangefinder, and it can rotate its two drive wheels. If the drive wheels (mounted on either side of the robot on a common axis) are rotated together in the same direction, the robot rolls forward or backward. If they are rotated in opposite directions, the robot turns about its center point. Thus, there are currently four robot activities that cause robot motion. Each has a single argument giving the magnitude of the motion:

<u>Activity</u>	<u>Argument</u>
TILT	Number of degrees upward from present position
PAN	Number of degrees right (clockwise) from present position
TURN	Number of degrees right (clockwise) from present position
ROLL	Number of feet forward.

Negative values of the arguments lead to directions of motion opposite to those listed above.

(The robot also has the ability to turn only one of its drive wheels, thus pivoting about the other. Since no actual demand to use this ability has arisen to date, the two corresponding activities are not currently provided.)

The robot presently has three sensory modes: TV, rangefinder, and catwhiskers. The first two of these can be activated on command. The command RANGE causes a reading of the distance to the nearest surface, along a path that is nominally in the center of the field of view of

the camera. The command SHOOT causes a quantized TV picture to be read into the PDP-10 memory. An auxiliary TV activity, TVMODE, is provided to enable setting of the picture resolution and (potentially) other aspects of the TV system, such as beam current and target voltage, or color filters in front of the camera. Two other auxiliary TV activities, IRIS and FOCUS, operate motors on the robot to control the iris setting (f-stop) and the distance of best focus of the camera.

The catwhiskers (and the push-bar) are affected only when the robot moves, through TURN or ROLL. At the conclusion of each such activity, a report of the status of these sensors is sent to the PDP-10 and stored. Thus, there is no activity corresponding directly to this mode of sensory input. However, an option exists as to whether changes in the status of the whiskers and push-bar during a TURN or ROLL cause the robot to halt immediately (the "normal" case), or whether the motion proceeds to completion (in which case we say the "overrides" are on). This option is controlled by activity OVRID. Details of the catwhisker operation are contained in a later section.

Thus, there are presently the following sensory-related activities:

<u>Activity</u>	<u>Argument</u>
RANGE	None
SHOOT	(Picture array location in PDP-10)
TVMODE	0: Set picture resolution to 120 x 120 1: Set picture resolution to 240 x 240
OVRID	0: Turn all overrides off 1: Turn on the catwhisker override only 2: Turn on the push-bar override only 3: Turn on both classes of override

<u>Activity</u>	<u>Argument</u>
IRIS	Number of exposure-value (EV) units by which to open up the iris (see below)
FOCUS	Number of feet by which the focal distance is to be increased.

In subsequent sections we will take a closer look at these activities. First, we must digress to consider some general characteristics of the bottom-level software package.

GENERAL CHARACTERISTICS AND DESIGN PHILOSOPHY

A call to one of the (LISP) functions implementing the activities listed above only starts that activity. That is, the called function causes the PDP-10 to communicate with the PDP-15 (through the routine START15), telling the PDP-15 to undertake the required action. Then the called function returns control to the program that called it. Thus, the robot program does not "hang up" in the called function while the activity is being carried out.

This design has several ramifications. First, it allows nonconflicting activities to be carried out concurrently. Conflicting activities are those for which the robot's actions literally interfere with each other (e.g., taking a TV picture while moving), or for which the maintenance of the robot's model would be garbled if one activity is not laid to rest before the other is begun. A table, included in this paper, shows the conflicting activities. The bottom-level software checks for conflicts and hangs up the robot program in an activity call until all conflicting activities from earlier calls are complete. There is no provision made for queuing such conflicting calls and allowing the robot program to proceed, since the need for this seemed too unlikely to warrant the effort.

Second, control may be almost anywhere in the robot program hierarchy when the previously requested activity terminates. Since the LISP system is not structured to allow arbitrary program interrupts, the robot program is not informed when an activity has terminated. The program has contact with the status of a previously requested activity only in the following instances:

- (1) The program calls an activity that conflicts with the previously called one
- (2) The program attempts to access information in the robot's model (using the N-tuple storage system routines whose names begin with M, for model), which information might be changed by the previously called activity.

In either of these instances, the bottom-level software automatically causes the PDP-10 to obtain from the PDP-15 a reading of the status of the previously called activity. If the activity has terminated, the software performs necessary bookkeeping and allows the new request to proceed. If not, the new request is hung up in a wait loop until a subsequent reading from the PDP-15 indicates that the former activity is terminated.

In any case, information about the status of the external activity does not even enter the PDP-10 until it specifically requests a reading from the PDP-15, no matter how far in the past the activity may have terminated. This is a consequence of the fact that the PDP-10 initiates all intercomputer transfers; it may be viewed as a scheme of receiving information from the PDP-15 and robot only on a "need-to-know" basis.

An important corollary of this design is that requests to the robot model to access information which may be affected by robot activities should always be made via the N-tuple storage system functions beginning with "M," not those beginning with "NT." Otherwise, the necessary interlocking will not occur, and obsolete information could be accessed.

THE ACTIVITY STATUS VARIABLE (ASV) AND
MODES OF TERMINATION

We have seen that information flows from the PDP-15 to the PDP-10 only when the PDP-10 requests a status report on an activity. The first data element of such a report is called the activity status variable (ASV). For the particular activity being interrogated, the ASV tells whether that activity has terminated, and, if so, in what way. If the activity is still in progress, the ASV has the value -1, and the remainder of the report is meaningless. If the activity has terminated, the ASV has a nonnegative value, which is subsequently available in the robot's model.

Various (nonnegative) values of the ASV have specific meanings for different activities, which will be described subsequently. Certain values, however, have meanings common to all the activities:

<u>ASV</u>	<u>Mode of Termination</u>
6	Time-out occurred
7	Activity was STOP-ped by PDP-10
8	Terminated by panic in PDP-15.

Time-outs are determined by the PDP-15. One of our design decisions was that every activity would terminate after some specified time, no matter what the condition of the robot or its communication link (assuming only proper operation of the PDP-15). Thus, the user can avoid the common frustration of having his program hang up on the external equipment and needing to restart it from scratch.

There is a provision (which we expect to be very rarely used) for the robot program, after starting an activity, to abort it (whether or

not it has terminated). This is done by invoking the STOP15 routine. On receiving the corresponding command, the PDP-15 halts the action of the robot. The subsequent status report (which, as always, does not go to the PDP-10 until requested) will have an ASV of 7. We will endeavor to provide, in such status reports, valid information on the terminal status of motor registers, etc., in the robot.

The panic ASV is a catch-all for reporting hardware or software malfunctions with which the PDP-15 cannot cope.

In what follows, an ASV value of 0 generally represents the most common or "most normal" mode of termination. The precise meaning of this and other ASV values, however, depends on the activity.

ADDITIONAL CHARACTERISTICS OF PAN AND TILT

Sometimes the user (by which we mean either a person at a Teletype or some higher-level program in the PDP-10) may want the robot to pan incrementally (i.e., turn its head x degrees left or right of where it is currently), and sometimes an absolute positioning is desired (x degrees left or right of the forward position). The activities specified in an earlier section are parameterized on an incremental basis, so the incremental case is handled directly. (For every statement made in this section about PAN, an analogous statement applies to TILT.)

For absolute positioning, let us first make the assumption that an entry in the robot's model in the PDP-10 contains the current value of the pan angle of the robot's head. Then a routine (call it PANTO) whose argument is an absolute pan angle, in degrees, can proceed as follows: first, PANTO accesses the model to determine the current pan angle; second, PANTO subtracts the current value from the desired one, to determine the necessary increment; third, PANTO calls PAN, specifying this increment. PAN causes the action to be performed, and updates the robot model.

Now consider the case in which the model is not assumed to have an accurate value of the pan angle. This case can arise in the start-up of an experiment, through malfunction or error, or as a result of steady accumulation of uncertainty. In this case a user can establish the pan position by first requesting a pan activity with an excessive increment (which will drive the pan mechanism against one of its limit switches), setting the pan angle to the value of that limit in the robot model, then performing a PAN or PANTO to the desired position. What is required

in addition to the basic PAN activity is knowledge about the limit value. Since this is a constant of the robot hardware, it can be determined and coded into the appropriate program or programs.

Zero for the pan coordinate occurs when the head assembly is facing straight ahead on the robot. Zero for the tilt coordinate occurs when the axis of the TV camera and rangefinder is horizontal.

The input parameter for the routine PAN is expressed in degrees (a floating-point number), as is the value of the pan angle in the robot model. For a communication to the PDP-15 and the robot, the pan angle increment is expressed in counts of the digital register that drives the pan motor on the robot. (This is a matter of removing all possible computational burdens from the PDP-15.) The bottom-level software performs the necessary conversion using a constant PANFACTOR = counts/degree.

TERMINATION MODES FOR PAN (AND TILT, BY ANALOGY)

ASV=0. The pan activity achieved the requested increment. The value of the pan angle in the model is updated at the time this status report is received.

ASV=1. The pan carriage ran into one of its limit switches. From knowledge of this fact and of the requested pan direction, the value of the pan angle in the model is updated to the limit value.

No other normal terminations of PAN are possible.

The same codes and analogous results apply to TILT.

THE CATWHISKERS

The catwhiskers are the tactile sensors of the robot. They consist of arcs of wire looping out from the robot's body and attached to micro-switches at both ends. A modest pressure on a whisker at almost any point will activate at least one of its two switches, which are arranged in parallel. In this event, we say that that whisker is "on."

Although the logic of the hardware catwhisker operation is complicated, there are only three cases that should be of interest to the user.

Case 1: At the beginning of a roll or turn activity, all catwhiskers are off and the catwhisker overrides are off. If nothing happens to the whiskers, the activity should go to completion. If a catwhisker is turned on by contacting some object during the activity, the robot will begin to decelerate. Then, if the catwhisker turns off (because it merely brushed an object) before the robot stops, the robot is supposed to pick up speed and complete its activity. Otherwise, the robot will quickly come to a stop. The robot will fall short of its desired position, unless it should happen to attain its goal while stopping.

Case 2: When a roll or turn is requested, a catwhisker is on and the overrides are off. In this case the robot will not move, and the action will be terminated.

Case 3: The overrides are on. Whether or not the catwhiskers are on or come on during the activity, the robot is not halted and should complete its activity.

The user receives, in the status report of a terminated roll or turn activity, three quantities: the ASV, the residual count in the wheel-motor register (zero if the desired position was achieved), and a word whose bits give the status of the catwhiskers on termination. From these values it is possible to reconstruct what happened.

TERMINATION MODES FOR ROLL (AND TURN, BY ANALOGY)

ASV=0. A full roll was completed. No catwhiskers came on. The residual count and the catwhisker word should be zero.

ASV=1. A full roll was completed. Catwhiskers came on, but were ignored. This implies that the catwhisker overrides were on. The residual count should be zero, and the catwhisker word should reflect whatever status the whiskers had on termination.

ASV=2. A full roll was not completed. A bump or bumps occurred, and the catwhisker override was off. The residual count is in general not zero. The catwhisker word reflects the terminal status. This outcome could arise from either Case 1 or Case 2 above.

ASV=3. A full roll was not completed, because the push-bar (see below) became free (and the push-bar override was off). Residual count and catwhisker word are as in ASV=2.

In the case of ASV=0, the user can conclude that the activity proceeded as intended. In other cases, it seems that the residual count and the terminal catwhisker word are more valuable than the ASV. In all cases, the bottom-level software updates the model with the new values of the robot's X and Y location and angular position, based on the old values, the requested move, and the residual count. The catwhisker status value in the model is also updated with the new terminal value.

THE PUSH-BAR

The robot is presently fitted with a push-bar on the front, with two switches. One switch is to tell when the bar is pushing against an object; the other, when the bar is encountering excess resistance from an unmoving object or a wall. The former signals the PDP-15 (i.e., generates a special interrupt) whenever it goes from on (contacted) to off (free). This is to tell the program when a pushed object has slipped off the bar. Normally, this will cause the PDP-15 to stop the robot; however, this can be overridden. The status of this switch on termination of an activity is reported as one of the bits in the catwhisker status word.

The second switch, signaling excess resistance, will cause the PDP-15 to execute and subsequently report an ASV=8 panic stop.

OVERRIDES

There are presently two overrides, one for the catwhiskers as a group and one for the first switch on the push-bar. These are entirely separate and operate somewhat differently. The catwhisker override, which goes to a hardware register on the robot, blocks the robot from executing its early shutdown sequence (see Case 1, above) that otherwise occurs whenever a catwhisker is on. The push-bar override blocks the PDP-15 from stopping the robot whenever the first push-bar switch makes a transition from on to off.

When the robot is pushing an object, the catwhisker override must be on. When the robot is backing off from an object, the catwhisker override must always be on and the push-bar override must be on if the robot is to back off beyond the point of disengagement in one motion.

Activity OVRID turns the overrides on and off according to the value of its argument, as follows:

<u>Argument</u>	<u>Catwhisker Override</u>	<u>Push-bar Override</u>
0	OFF	OFF
1	ON	OFF
2	OFF	ON
3	ON	ON

Being an activity, OVRID is subject to the ASV discipline. A program that calls OVRID and then, say, TURN or ROLL may have to wait momentarily for the termination of OVRID. On termination, the only ASV's that are likely to occur are ASV=0 (completed) or ASV=8 (panic; probably transmission error).

ADDITIONAL CHARACTERISTICS OF TURN AND ROLL

The input parameter for ROLL is expressed in feet; that for TURN, in degrees. As with PAN and TILT, the arguments are converted to motor counts for communication with the PDP-15. Motion is inherently incremental, there being no limit switches involved and no absolute knowledge of position except what can be deduced by the robot program. Whenever the robot executes TURN, the bottom-level software updates its angular position, θ , in the model. When the robot executes ROLL, the software updates the robot's location:

$$X - X' + (\cos \theta) \cdot (\text{distance moved})$$

$$Y - Y' + (\sin \theta) \cdot (\text{distance moved}) \quad .$$

It is implicit that the current X, Y, and θ are always available in the model, although they are subject to revision by higher authority (i.e., the user) at any time.

RANGE AND SHOOT

Activity RANGE has no arguments. Activity SHOOT may have as an argument the location of an array in the PDP-10 to receive the TV picture. The terminal ASV's that are to be expected are ASV=0 (completed), ASV=6 (timeout), and ASV=8 (panic).

Both of these activities require a turn-on time measured in seconds, to bring the rangefinder mirror up to speed in one case and to warm up the TV electronics in the other. (These and other operating modules in the robot are normally kept off to conserve power.) To avoid waiting for the turn-on every time during a period of repetitive use, without burdening the user with predicting such periods, we have established time-outs for these modules, controlled by the PDP-15. After RANGE or SHOOT, the corresponding module will be kept on for a period of, say, a minute. When this time has passed since the last such activity, the module will be shut down. This will be done without effort on the user's part.

TVMODE, IRIS, AND FOCUS

These activities are used to prepare for taking a TV picture. They may be called at any time.

Activity TVMODE prepares for 120×120 pictures to be read subsequently if its argument is zero, and for 240×240 pictures if its argument is 1. Like OVRID, TVMODE has an ASV whose terminal value should only be 0 or 8.

IRIS and FOCUS operate motors with limit switches. In most respects, these activities are analogous to PAN and TILT. IRIS and FOCUS, however, perform a nonlinear transformation between the input argument and the drive-motor count. Multiplying the desired increment by a constant factor does not suffice; the true transformation is determined by calibration and stored within the bottom-level software.

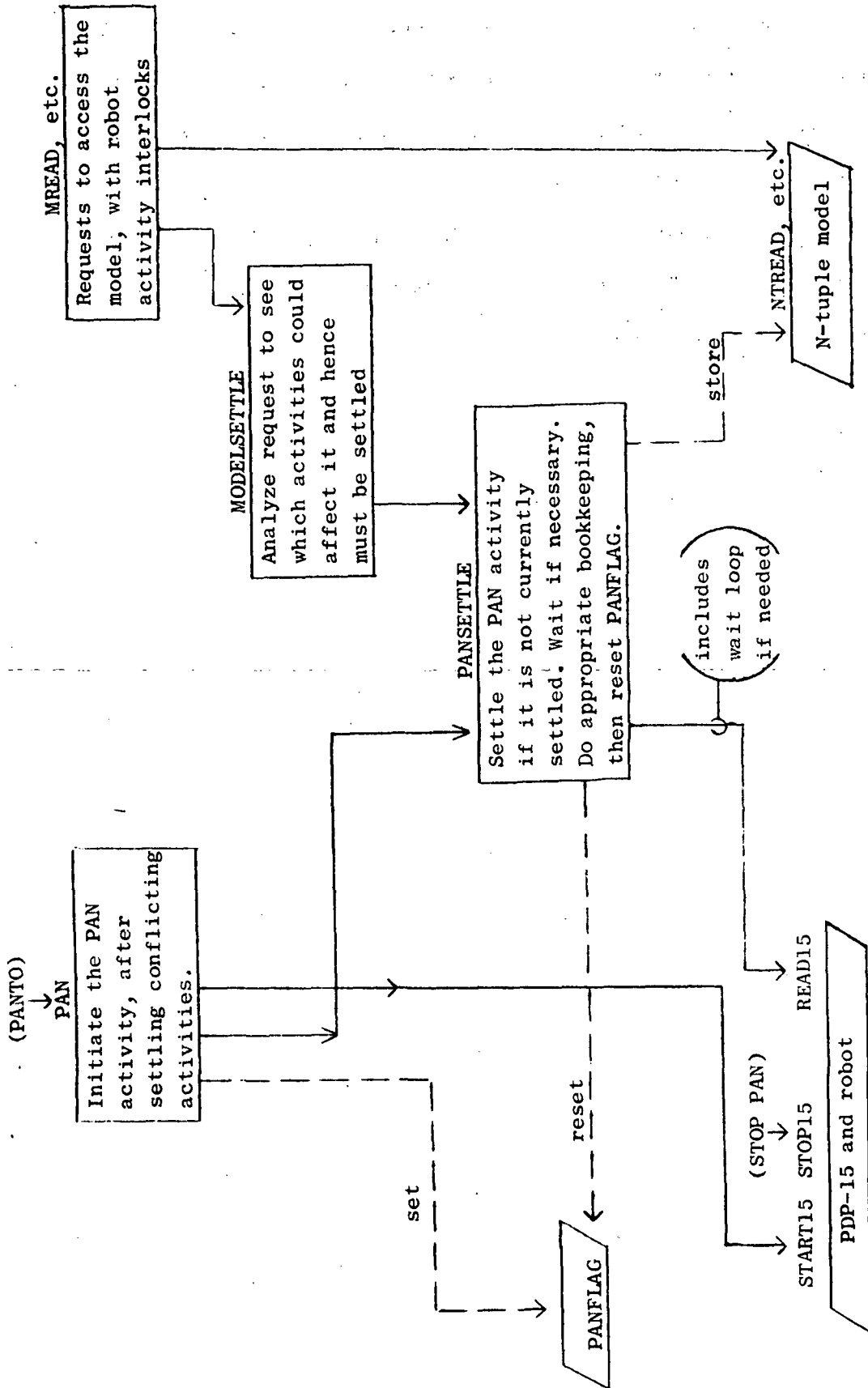
The argument for IRIS is expressed as an increment in the exposure value (EV), which is logarithmically related to the f-value of the iris opening. No matter what the current EV number, increasing it by one doubles the light reaching the camera.

The argument for FOCUS is expressed as an increment, in feet, by which the distance to the plane of best focus (f) is to be increased. This form of the argument is chosen because it is most convenient for the user, but it leads to a problem for the software. The motion of the focus drive motor is highly nonlinear in f , being more nearly linear in $1/f$. A tiny motion of the drive motor carries f all the way from, say, 100 feet to infinity.

We thus establish the following convention for the argument of FOCUS. Any increment that carries f beyond 100 feet is treated

by driving the focus carriage to its outer limit (i.e., focus at infinity), and the value of f in the robot model is set to 100 feet. Of course, any subsequent increment that reduces the focal distance will be treated relative to an initial f of 100 feet.

The user can avoid any concern with this issue by using a routine FOCUSTO, analogous to PANTO, that takes an absolute distance argument rather than an incremental one.



Note: The structure is illustrated for the PAN routines. Duplicate routines exist for other activities.

PROGRAM STRUCTURE OF BOTTOM-LEVEL

PDP-10 ROBOT SOFTWARE

Name	TILT	PAN	TURN	ROLL	OVRID
Activity code	1	2	3	4	5
Argument	Degrees up	Degrees right	Degrees right	Feet forward	0: off. 1: whiskers. 2: bar. 3: both
ASV (6=time out; 7=stop; 8=panic)	0: complete 1: limit	0: complete 1: limit	0: complete; 1: bumped, ignored; 2: bumped, stopped;	0: bumped, ignored; 3: push-bar	0: complete
Second element of PDP-15 status report	residual motor count	residual motor count	residual motor count	residual motor count	--
Third element	--	--	Terminal status of whiskers		--
Conflicting activities	TILT, RANGE, SHOOT	PAN, RANGE, SHOOT	TURN, ROLL, OVRID, RANGE, SHOOT	TURN, ROLL, OVRID, RANGE, SHOOT	TURN, ROLL, OVRID
Information updated in model (incl. ASV)	tilt angle	pan angle	theta, whiskers	X, Y, whiskers (θ is accessed)	overrides
Constants required	high limit low limit counts/deg.	left limit right limit counts/deg.	counts/deg.	counts/foot	--

Form of catwhisker
status word:

Name	RANGE	SHOOT	TVMODE	IRIS	FOCUS
Activity code	6	7	8	9	10
Argument	--	(picture array address)	0: 120 X 120 1: 240 X 240	EV units toward "open"	feet farther from robot
ASV (6=time out; 7=stop; 8=panic)	0: complete	0: complete 1: Xmission error	0: complete	0: complete 1: limit	0: complete 1: limit
Second element of PDP-15 status report	range register count	--	--	residual motor count	residual motor count
Third element	--	--	--	--	--
Conflicting activities	TILT, PAN, TURN, ROLL, RANGE	all except OVRID & RANGE	SHOOT, TVMODE	SHOOT, IRIS	SHOOT, FOCUS
Information updated in model (incl. ASV)	"last range"	"last picture"	mode setting	IRIS value	FOCUS distance
Constants required	triangulation function	--	--	shut limit open limit counts/EV	near limit far limit = 100' conversion fn.

APPENDIX C

THE FRAME PROBLEM IN PROBLEM-SOLVING SYSTEMS

June 1970

THE FRAME PROBLEM IN PROBLEM-SOLVING SYSTEMS

by

Bertram Raphael

Paper prepared for publication in the Proceedings of the Advanced Study Institute on Artificial Intelligence and Heuristic Programming, held at Menaggio, Italy, August 1970.

Artificial Intelligence Group

Technical Note 33

SRI Project 8259

This research is sponsored by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS 12-2221.

I INTRODUCTION

The frame problem has taken on new significance during recent attempts to develop artificially intelligent systems. The problem deals with the difficulty of creating and maintaining an appropriate informational context or "frame of reference" at each stage in certain problem-solving processes. Since this is an area of current research, we are not prepared to present a solution to the frame problem; rather, the purpose of this paper is to sketch the approaches being pursued, and to invite the reader to suggest additions and improvements.

Although broader interpretations are possible, we think of an "artificially intelligent system" as meaning a programmed computer, with associated electronic and mechanical devices (e.g., a radio-controlled robot vehicle and camera), that is intelligent in the sense defined by McCarthy and Hayes: (1)*

"... we shall say that an entity is intelligent if it has an adequate model of the world (including the intellectual world of mathematics, understanding of its own goals and other mental processes), if it is clever enough to answer a wide variety of questions on the basis of this model, if it can get additional information from the external world when it wants to, and can perform such tasks in the external world that its goals demand and its physical abilities permit."

Reference (2) discusses the research significance of attempting to build such an intelligent robot system.

* References are listed at the end of this paper.

The intelligent entity, as defined above, will have to be able to carry out tasks. Since a task generally involves some change in the world, it must be able to update its model so that it remains as accurate during and after the performance of a task as it was before. Moreover, it must be able to plan how to carry out a task, and this planning process usually requires keeping "in mind," simultaneously, a variety of possible actions and corresponding models of the hypothetical worlds that would result from those actions. The bookkeeping problems involved with keeping track of these hypothetical worlds account for much of the difficulty of the frame problem.

II THE FRAME PROBLEM

We shall illustrate the frame problem with a simple example. Suppose the initial world description contains the following facts (expressed in some suitable representation, whose precise form is beyond our immediate concern):

- (F1) A robot is at position A.
- (F2) A box called B1 is at position B.
- (F3) A box called B2 is on top of B1.
- (F4) A,B,C, and D are all positions in the same room.

Suppose, further, that two kinds of actions are possible:

- (A1) The robot goes from x to y, and
- (A2) The robot pushes B1 from x to y,

where x and y are in {A,B,C,D}. Now consider the following possible tasks:

Task (1): The robot should be at C.

This can be accomplished by the action of type A1, "Go from A to C."

After performing the action, the system should "know" that facts F2

through F4 are still true, i.e. they describe the world after the action, but F1 must be replaced by

(F1') The robot is at position C.

Task (2): B1 should be at C.

Now a "push" action must be used, and both F1 and F2 must be changed.

One can think of simple procedures for making appropriate changes in the model, but they all seem to break down in more complicated cases.

For example, suppose the procedure is:

Procedure (a): "Determine which facts change by matching the task specification against the initial model."

This would fail in task (1) if the problem solver decided to get the robot to C by pushing B1 there (which is not unreasonable if the box were between the robot and C and pushing were easier than going around), thus changing F2.

Procedure (b): "Specify which facts are changed by each action operator."

This procedure is also not sufficient, for the initial world description may also contain derived information such as

(F6) B2 is at position B,

which happens to be made false in task (2).

More complicated problems arise when sequences of actions are required. Consider:

Task (3): The robot should be at D and, simultaneously, B2 should be at C.

The solution requires two actions, "Push B1 from B to C" and "Go from C to D," in that order. Any effective problem solver must have access to

the full sets of facts, including derived consequences that will be true as a result of each possible action, in order to produce the correct sequence.

Note that the frame problem is a problem of finding a practical solution, not merely finding a solution. Thus it resembles the famous traveling salesman problem or the problem of finding a winning move in a chess game; problems for which straightforward algorithms are known but usually worthless.

McCarthy and Hayes⁽¹⁾ divide intelligence into two parts: the epistemological part, which deals with the nature of the representation of the world, and the heuristic part, which deals with the problem-solving mechanisms that operate on the representation. They then proceed to concentrate upon the epistemological questions related to several aspects of intelligence (including the frame problem). Here, on the other hand, we are concerned with constructing a complete intelligent system, including both the world representations and the closely related problem-solving programs. In the following we shall assume that the representations are basically in the form preferred in Ref. (1), namely sets of sentences in a suitable formal logical language such as predicate-calculus; and we shall describe candidate organizations for the "heuristic part," i.e. the problem solver, of an artificially intelligent system that can cope with the frame problem.

III CURRENT APPROACHES

A. Complete Frame Descriptions

A frame can generally be completely described by some data structure, e.g. by a set of facts--expressed as statements in a predicate

calculus. If we think of each such frame as an object and each possible action as an operator that can transform one object (frame) into another, then we may use a problem-solving system such as GPS⁽³⁾ for attempting to construct an object for which the desired goal conditions are true. Unfortunately, when the data base defining each frame reaches a non-trivial size, it becomes impractical to generate and store all the complete frame-objects. For example, suppose each possible frame is defined by 1000 elementary facts, an average of six different actions are applicable and heuristically plausible in any situation, and a typical task requires a sequence of four actions--not unreasonable assumptions about a simple robot system. Then the search tree of possible frames may have about 1000 nodes; it is not practical to store 1000 facts at each node. If each action causes changes in, say, three facts, then storing just the change information at each node is practical--provided appropriate bookkeeping is done to keep track of which of the original facts still holds after a series of actions. This bookkeeping seems to require considerable program structure in addition to (and quite separate from) the basic object, operator, and difference structure of a GPS-type system. The following approaches are concerned with this new bookkeeping problem.

B. State Variables

One way to keep track of frames is to consider each possible world to be in a separate state and to assign names to states. In this formulation, actions are state transition rules, i.e. rules for transforming one state into another. Since action rules are generally applicable to large classes of states, the description of an action can contain variables that range over state names.

Green describes an approach of this kind in detail in Ref. (4). Each fact is labeled with the name of the state in which it is known to be true. Additional facts that are state-independent describe the transitional effects of actions. For example, if S_0 is the name of the initial state and $At(ob, pos, s)$ is a predicate asserting that object ob is at position pos in state s, then the conditions of the previous example may be partially defined by the following axioms:

- (G1) $At(Robot, A, S_0)$ (from F1)
- (G2) $At(B1, B, S_0)$ (from F2)
- (G3) $Box(B1) \wedge Box(B2)$ (B1 and B2 are boxes)
- (G4) $(\forall x, y, x) [At(Robot, x, s) \supset At(Robot, y, go(x, y, s))]$ (from A1)

At this point some explanations seem in order. $Box(x)$ asserts that x is a box. Perhaps it would have been more consistent to write, -e.g., $Box(B1, S_0)$, because we only know that B1 is a box in the initial state. However, we do not contemplate allowing any actions that destroy box-ness, such as sawing or burning, so we could add the axiom $(\forall s)Box(B1, s)$. Since we would then be able to prove that B1 is a box in all states, we suppress the state variable without loss of generality.

Each action, in this formalism, is viewed as a function. One argument of the function is always the state in which the action is applied, and the value of the function is the state resulting after the action. Thus, e.g., the value of $go(A, C, S_0)$ is the name of the state achieved by going to C after starting from A in the initial state.

The appeal of this approach is that, if we have a theorem-proving program, no special problem-solving mechanisms or bookkeeping procedures are necessary. Action operators may be fully described by ordinary axioms

(such as G4 for the go operation) and the theorem-proving program, with its built-in bookkeeping, becomes the problem solver. For example, task (1) may be stated in the form, "Prove that there exists a state in which the robot is at C," or in predicate calculus, prove the theorem:

$$(*) \quad (\exists s) \text{ At}(\text{Robot}, C, s) \quad .$$

From (G1) and (Gr), we can prove that (*) is indeed a theorem. By answer tracing during the proof (Ref. (5)), we can show that $s = \text{go}(A, C, S_0)$, which is the solution.

For more complex actions, however, the major problem with this approach emerges: After each state change, the entire data base must be reestablished. We need additional axioms that tell not only what things change with each action, but also what things remain the same. For example, we know that B1 is at B in state S_0 (by G2), but as soon as the robot moves, say to state $\text{go}(A, C, S_0)$, we no longer know where B1 is! To be able to figure this out, we need another axiom, such as

$$(\forall x, y, u, v, s) [\text{At}(x, y, s) \wedge x \neq \text{ROBOT} \supset \text{At}(x, y, \text{go}(u, v, s))] \quad .$$

("When the robot goes from u to v, the object x remains where it is at y.") Thus a prodigious set of axioms is needed to define explicitly how every action affects every predicate, and considerable theorem-proving effort is needed to "drag along" unaffected facts through state transitions. Clearly this approach will not be practical for problems involving many facts.

C. The World Predicate (6)

Instead of using a variety of independent facts to represent knowledge about a state of the world, suppose we take all the facts about a particular world and view the entire collection as a single entity, the

model \mathfrak{m} . We may then use a single predicate P , the "world predicate," whose domains are models and state-names, $P(\mathfrak{m},s)$ is interpreted as meaning that s is the name of the world that satisfies all the facts in \mathfrak{m} . One possible structure for \mathfrak{m} is a set of ordered n -tuples, each of which represents some elementary relation; e.g., $\langle \text{At}, \text{Robot}, A \rangle$ and $\langle \text{At}, B1, B \rangle$ are elements of the initial model, \mathfrak{m}_1 .

The initial world is defined by the axiom $P(\mathfrak{m}_1, S_0)$ (except that the complete known contents of \mathfrak{m}_1 must be explicitly given). We can now specify that an action changes a particular relation in \mathfrak{m} , and does not change any other relations, by a single axiom, e.g. the go action is defined by the axiom

$$(\forall x,y,\bar{w},s) [P(\{\langle \text{At}, \text{Robot}, x \rangle, \bar{w} \}, s) \supset P(\{\langle \text{At}, \text{Robot}, y \rangle, \bar{w} \}, \text{go}(x,y,s))] .$$

Here \bar{w} (read "w-bar") is a variable whose value is an indefinite number of elements of a set, namely all those that are not explicitly described.

This approach preserves the advantages of the previous state-variable approach; namely, the problem solving, answer construction, and other bookkeeping can be left to the theorem prover. In addition, properties of the model are automatically carried through state changes by the barred variables. On the other hand, several difficulties are apparent: theorem-proving strategies may be grossly inefficient in the domain of problem solving; the logic must be extended to include domains of sets and n -tuples; complex pattern-matching algorithms will be needed to compare expressions containing variables that range over individuals, n -tuples, sets, and indefinite subsets; and the fact that properties of the world are stored as data, instead of as axioms, constrains the problem-solving process by restricting the class of inferences that are possible. Further study is necessary to determine the feasibility of this approach.

D. Contexts and Context Graphs ⁽⁷⁾

Suppose we let a state correspond to our intuitive notion of a complete physical situation. Since the domain of our logical formalism includes physical measurements such as object positions, descriptions, etc., every consistent statement of first-order logic is either true or false for every state. We think of each such statement as a predicate that defines a set of states, namely those for which it is true. We call such a set of possible states the context defined by the predicate.

We shall find it convenient to allow certain distinguished variables, called parameters, to occur in predicates. Since each such predicate with ground terms substituted for parameters defines a context, a predicate containing parameters may be thought of as defining a family of possible contexts--and each partial instantiation of parameters in the predicate defines a subfamily of contexts (or, if no parameters remain, a specific context).

For example, the predicate $At(B1,B)$ defines a context (the set of all states) in which object B1 is at position B. If x and y are parameters, $At(x,y)$ defines the family of contexts in which some object is located any place. $At(B1,y)$ is a subfamily of this family in which the object B1 must be located at some (as yet unspecified) place.

A problem to be solved is specified by a particular predicate called the goal predicate. The problem, implicitly, is to achieve a goal state, i.e., produce any member of the context defined by the goal predicate.

An action will consist of an operator name, a parameter list, and two predicates--the preconditions K and the results R . In addition,

any of the elementary relations in the preconditions may be designated as transient preconditions. For example, the go action is defined by

$$\begin{array}{c} \text{name parameters} \\ \underbrace{\text{go}} \quad \underbrace{(x,y)} \\ K\{\underline{\text{At(Robot,x)}} \mid \text{At(Robot,y)}\}R \end{array} ,$$

where underlining designates a transient condition. Each action operator thus corresponds to a family of specific actions. An action is applicable in any state that satisfies K ; when an action is applied, the resulting state no longer need satisfy the transients but must satisfy R .

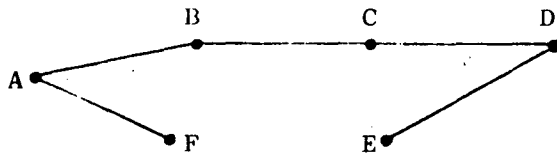
In this approach, the conjunction of predicates in the robot's model of the world is an initial predicate I , defining as an initial context the set of all states that have, in common, all the known properties of the robot's current world. The goal context, defined by a given goal predicate, is the set of satisfactory target states. When an operator is applied in a context, it changes the defining predicate (roughly, by deleting transients and conjoining results), thereby changing the context. The problem-solving task is to construct a sequence of operators that will transform the initial context into a subset of the goal context.

Any context that can be reached from the initial context by a finite sequence of operators is called an achievable context. Any context from which a subset of the goal context can be reached by a finite sequence of operators is called a sufficient context. The main task may be restated, then, as finding an operator sequence to show that the goal is achievable, or that the initial context is sufficient, or, more generally, that some achievable context is a subset of some sufficient context (and therefore is itself sufficient).

The main loop of the problem solver consists of two steps:

- (1) Test whether any known achievable context is a subset of any known sufficient context. If so, we are done.
- (2) Either generate a new achievable context by applying some operator in a known achievable context ("working forwards"), or generate, as a new sufficient context, one that would become a known sufficient context by the application of some operator ("working backwards"). Then return to step 1 to test the newly generated context.

An advantage of this approach is that all states and all properties of operators are defined by first-order predicates, so a standard theorem-proving program can do most of the work of testing operators and results and selecting values of parameters. On the other hand, a separate data structure, called a context graph, is needed to keep track of the trees of achievable and sufficient states and the operators that relate their nodes. For example, suppose we wish to get from A to D in the directed graph:



We shall abbreviate by \mathcal{G} the predicate that gives the graph's topology:

$$\mathcal{G} \equiv \text{Path}(A,B) \wedge \text{Path}(B,C) \wedge \text{Path}(C,D) \wedge \text{Path}(A,F) \wedge \text{Path}(E,D)$$

The initial predicate is $I \equiv \text{At}(A) \wedge \mathcal{G}$. The goal predicate is $G \equiv \text{At}(D)$.

We shall define the operator go, for this problem, by:

go(x,y)	
Path(x,y)	At(y)
<u>At(x)</u>	

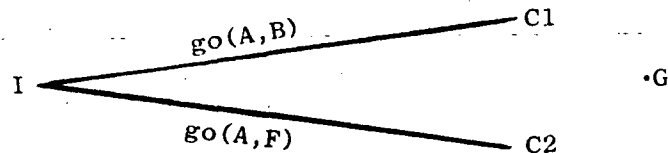
The operator is applicable in context I only if we can prove that

$$(\forall x,y) [I \supset At(x) \wedge Path(x,y)]$$

is a theorem. The proof can be done by resolution with answer tracing.⁽⁵⁾

The above statement can be shown to be a theorem when $x = A$ and y is either B or F. Therefore, the go operator can be used two ways to generate new achievable contexts C1 and C2, with corresponding predicates

$P_{C1} = \mathcal{J} \wedge At(B)$, $P_{C2} = \mathcal{J} \wedge At(F)$. To keep track of actions and instantiations, we shall draw the context graph:



Similarly, from C1 we can prove the applicability of $go(B,C)$, which, when applied, gives C3

$$P_{C3} = \mathcal{J} \wedge At(C)$$

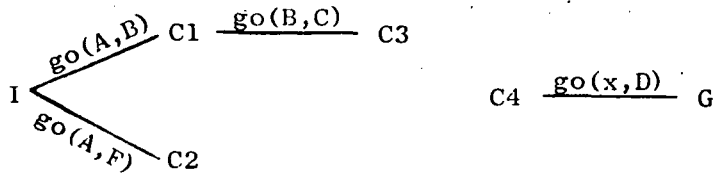
To illustrate working backwards, consider whether the result of a go implies G. The relevant problem for a theorem prover is

$$(\exists y) [At(y) \supset At(D)]$$

This is trivially true if $y = D$, so any state that satisfies the preconditions of the operator $go(x,D)$ is sufficient (because the operator will then be applicable, and will produce the goal). Thus a new sufficient context is given by the preconditions,

$$P_{C4} = At(x) \wedge Path(x,D)$$

(Note that C4 is really a family of contexts, because of the parameter x.) The context graph is now:



Finally, the theorem prover can show that

$$P_{C3} \supset P_{C4} \text{ when } x = C \text{ ,}$$

completing the solution.

Most problems are considerably more difficult than the above example because of several complications. Suppose in trying to work backwards from G (using an operator Op with preconditions K and results R) we find that we cannot prove $R \supset G$, but instead discover a statement S such that $R \wedge S \supset G$. We may still work backwards with Op, but the new sufficient context is defined not by K alone but rather by $K \wedge S$. Furthermore, some extra bookkeeping must remind us that S may not be disturbed, in a valid solution, by applying Op--e.g., no transients of Op may appear in S. Similar additional subgoals--and bookkeeping complications--arise from each incomplete attempt to prove that an achievable context is contained in a sufficient context.

Additional complexities arise from dependencies. That is, when an expression E is deleted by a transient during an action, other expressions that were deduced from E in previous contexts can no longer be guaranteed to be true in new contexts. Thus each deduced expression is said to depend upon all its ancestors, adding to our growing burden of bookkeeping problems.

On the other hand, the context graph can take care of much of the bookkeeping automatically. Each logical expression need only be stored once, with notations telling in which contexts it was created and destroyed, rather than being either copied or rederived from context to context. Finally, if predicates of achievable contexts and operator results are stored in clause form, and predicates of sufficient contexts and operator preconditions are stored in negated clause form, preliminary experiments show that most of the nuts-and-bolts work of attempting solutions and generating new contexts can be done in a straightforward manner by an existing resolution-type theorem-proving program.

E. Other Approaches

Several other approaches to the frame problem have been suggested, although few have been worked out in sufficient detail to test on a computer.

Richard E. Fikes at SRI is developing a system whose formal framework is similar to that of D above ("contexts and context graphs"), but which does not use resolution techniques. Instead, proofs are strongly dependent upon the semantics of the logic, and the problem solver proceeds by a heuristic, goal-directed, case-analysis approach. This work is still in an early stage of development.

Eric Sandewall at Stanford is extending some ideas suggested by John McCarthy for formalizing the concepts of causality and time dependence, using a method proposed by J. Alan Robinson for embedding higher order logic in first-order predicate calculus. The resulting system provides an interesting model for inevitable sequences of events

(e.g. "if it is raining then things will get wet,") but may not be as useful for describing alternative possible actions by an external agent (e.g., the robot).

Methods for proving theorems in higher order predicate calculus are being developed in several places, and the use of this more powerful formalism may eventually vastly simplify our tasks. Finally, McCarthy and Hayes⁽¹⁾ suggest some other approaches including modal logics and counterfactuals, but the details have not been extensively explored.

IV CONCLUSIONS

This paper has described the frame problem and the principal methods that have been proposed for solving it.

Let us review the approaches listed above. A, complete frame and frame-transition descriptions, was simply a stage-setting "straw man" that we would not consider actually using. B, the logic-cum-state-variable approach, is beautifully elegant for "toy" problems, but both the representational effort and the theorem-proving effort grow explosively with problem complexity. C, the world predicate idea, preserves some of the elegance of approach B while carrying along necessary frame information implicitly; however, it places a burden on theorem-proving abilities in new domains and requires an awkward use of two levels of logical representation (that is, relations among the n-tuples in the model must be defined in terms of the world predicate), so that the practicality of the approach is open to serious question. Approach D, the use of contexts and context graphs (without explicit state names in the logic), is a more-or-less brute-force attempt to combine the use of first-order theorem-proving methods with a GPS-like structure of subgoals and operators;

although the bookkeeping problems are complicated, they seem to be tractable, so that the approach is reasonably promising. Finally, under E we mentioned several interesting ideas that warrant further exploration before they can be meaningfully compared with the other approaches.

Until now most research in problem solving has dealt with fairly static situations in narrow subject domains. As we become interested in building complete artificially independent systems, a new kind of problem-solving research emerges: We must study how to solve problems in an environment containing a large store of knowledge, while considering the possible effects of a variety of sequences of actions. This paper has described some of the first exploratory steps into this important area of research.

V ACKNOWLEDGMENT

Many of the ideas discussed above, including development of the contexts and context graphs approach, are the result of joint discussions between the author and L. S. Coles, R. E. Fikes, J. H. Munson, and N. J. Nilsson. The "world predicate" idea is completely due to R. Waldinger. C. C. Green developed the state variable approach, and he is largely responsible for our early realization of the importance of the frame problem.

The research described herein is supported by the Advanced Research Projects Agency of the Department of Defense, and by the National Aeronautics and Space Administration under Contract NAS12-2221.

REFERENCES

1. McCarthy and Hayes, "Some Philosophical Problems from the Standpoint of AI," Machine Intelligence 4, B. Meltzer and D. Michie, eds. (Edinburgh University Press, Edinburgh, Scotland, 1969).
2. B. Raphael, "The Relevance of Robot Research to AI," in Formal Systems and Non-Numeric Problem Solving by Computer (Springer-Verlag, 1970).
3. G. W. Ernst and A. Newell, GPS: A Case Study in Generality and Problem Solving (Academic Press, New York, N.Y., 1969).
4. C. C. Green, "Application of Theorem Proving to Problem Solving," Proc. International Joint Conference on Artificial Intelligence, Washington, D.C., May 7-9, 1969.
5. C. Green and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. 1968 ACM Conference, Las Vegas, Nevada (August 1968).
6. R. Waldinger, "Robot and State Variable," TN 26, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, California (April 1970).
7. B. Raphael, "Robot Problem Solving without State Variables," TN 30, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, California (May 1970).

APPENDIX D

STRIPS: A NEW APPROACH TO THE APPLICATION OF
THEOREM PROVING TO PROBLEM SOLVING

Page Intentionally Left Blank

Page Intentionally Left Blank

I INTRODUCTION

A. Overview of STRIPS

This note describes a new problem-solving program called STRIPS (Stanford Research Institute Problem Solver). The program is now being implemented in LISP on a PDP-10 to be used in conjunction with robot research at SRI. Even though the implementation of STRIPS is not yet complete, it seems to us important to discuss some of its planned features so that they can be compared with other on-going work in this area.

STRIPS belongs to the class of problem solvers that search a space of "world models" to find one in which a given goal is achieved. For any world model, we assume there exists a set of applicable operators each of which transforms the world model to some other world model. The task of the problem solver is to find some composition of operators that transforms a given initial world model into one that satisfies some particular goal condition.

This framework for problem solving, discussed at length by Nilsson,^{1*} has been central to much of the research in Artificial Intelligence. A wide variety of different kinds of problems can be posed in this framework.[†] Our primary interest here is in the class of

* References are listed at the end of this technical note.

† It is true that many problems do not require search and that specialized programs can be written to solve them. Our view is that these special programs belong to the class of available operators and that a search-based approach can be used to discover how these and other operators can be chained together to solve even more difficult problems.

problems faced by a robot in rearranging objects and in navigating. The robot problems we have in mind are of the sort that require quite complex and general world models compared to those needed in the solution of puzzles and games. Usually in puzzles and games, a simple matrix or list structure is adequate to represent a state of the problem. The world model for a robot problem solver, however, needs to include a large number of facts and relations dealing with the position of the robot and the positions and attributes of various objects, open spaces, and boundaries.

Thus, the first question facing the designer of a robot problem solver is how to represent the world model. A convenient answer is to let the world model take the form of statements in some sort of general logical formalism. For STRIPS we have chosen the first-order, predicate calculus mainly because of the existence of computer programs for finding proofs in this system. Initially, STRIPS will use the QA3 theorem-proving system² as its primary deductive mechanism.

Goals (and subgoals) for STRIPS will be stated as first-order predicate calculus wffs (well formed formulas). For example, the task "push a box to place b" might be stated as the wff $(\exists u)[\text{BOX}(u) \wedge \text{AT}(u,b)]$, where the predicates have the obvious interpretation. The task of the system is to find a sequence of operators that will produce a world model in which the goal can be shown to be true. The QA3 theorem prover will be used to determine whether or not a wff corresponding to a goal or subgoal is a theorem in a given world model.

Although theorem-proving methods will play an important role in STRIPS, they will not be used as the primary search

mechanism. A graph of world models (actually a tree) will be generated by a search process that can best be described as GPS-like (Ernst and Newell³). Thus it is fair to say that STRIPS is a combination of GPS and formal theorem-proving methods. This combination allows objects (world models) that can be much more complex and general than any of those used in previously implemented versions of GPS. This use of world models consisting of sets of logical statements causes some special problems that are now the subject of much research in Artificial Intelligence. In the next and following sections we will describe some of these problems and the particular solutions to them that STRIPS employs.

B. The Frame Problem

When sets of logical statements are used as world models, we must have some deductive mechanism that allows us to tell whether or not a given model satisfies the goal or satisfies the applicability conditions of various operators. Green⁴ implemented a problem-solving system based on a theorem prover using the resolution principle. In his system, Green expressed the results of operators as logical statements. Thus, for example, to describe an operator goto(x,y) whose effect is to move a robot from any place x to any other place y, Green would use the wff

$$(\forall x,y,s) [ATR(x,s) \Rightarrow ATR(y, goto'(x,y,s))] ,$$

where ATR is a predicate describing the robot's position. Here, each predicate has a state term that names the world model to which the predicate applies. Our wff above states that for all places x and y and for

all states s , if the robot is at x in state s then the robot will be at y in the state $\text{goto}'(x,y,s)$ resulting from applying the goto operator to state s .

With Green's formulation, any problem can be posed as a theorem to be proved. The theorem will have an existentially quantified state term, s . For example, the problem of pushing a box B to place b can be stated as the wff

$$(\exists s) AT(B,b,s) \quad .$$

If a constructive proof procedure is used, an instance of the state proved to exist can be extracted from the proof (Green,² Luckham and Nilsson⁵). This instance, in the form of a composition of operator functions acting on the initial state, then serves as a solution to the problem.

Green's formulation has all the appeal (and limitations) of any general-purpose problem solver and represents a significant step in the development of these systems. It does, however, suffer from some serious disadvantages that our present system attempts to overcome. One difficulty is caused by the fact that Green's system combines two essentially different kinds of searches into a single search for a proof of the theorem representing the goal. One of these searches is in a space of world models; this search proceeds by applying operators to these models to produce new models. The second type of search concerns finding a proof that a given world model satisfies the goal theorem or the applicability conditions of a given operator. Searches of this type proceed by applying rules of inference to wffs within a world model.

When these two kinds of searches are combined in the largely syntactically guided proof-finding mechanism of a general theorem prover, the result is gross inefficiency. Furthermore, it is much more difficult to apply any available semantic information in the combined search process.

The second drawback of Green's system is even more serious. The system must explicitly describe, by special axioms, those relations not affected by each of the operators. For example, since typically the positions of objects do not change when a robot moves, we must include the statement

$$(\forall u,x,y,z,s) [\text{OBJECT}(u,s) \wedge \text{AT}(u,x,s) \Rightarrow \text{AT}(u,x,\text{goto}'(y,z,s))] \quad .$$

Thus, after every application of goto in the search for a solution, we may need to prove that a given object B remains in the same position in the new state if the position of B is important to the completion of the solution.

The problem posed by the evident fact that operators affect certain relations and don't affect others is sometimes called the frame problem.^{6,7} Since, typically, most of the wffs in a world model will not be affected by an operator application, our approach will be to name only those relations that are affected by an operator and to assume that the unnamed relations remain valid in the new world model. Since proving that certain relations are still satisfied in successor states is tedious, our convention can drastically decrease the search effort required.

Because we are adopting special conventions about what happens to the wffs in a world model when an operator is applied, we have chosen

to take the process of operator application out of the formal deductive system entirely. In our approach, when an operator is applied to a world model, the computation of the new world model is done by a special extra-logical mechanism. Theorem-proving methods are used only within a given world model to answer questions about it concerning which operators are applicable and whether or not the goal has been satisfied. By separating the theorem proving that occurs within a world model from the search through the space of models we can employ separate strategies for these two activities and thereby improve the overall performance of the system.

II OPERATOR DESCRIPTIONS AND APPLICATIONS

The operators are the basic elements out of which a solution is built. For robot-like problems we can imagine that the operators correspond to routines or subprograms whose execution causes a robot to take certain actions. For example, we might have routines that cause the robot to turn and move, a routine that causes it to go through a doorway, a routine that causes it to push a box and perhaps dozens of others. When we discuss the application of problem-solving techniques to robot problems, the reader should keep in mind the distinction between an operator and its associated routines. Execution of routines actually causes the robot to take actions. Application of operators to world models occurs during the planning (i.e., problem solving) phase when an attempt is being made to find a sequence of operators whose associated routines will produce a desired state of the world. Since routines are

programs, they can have parameters that are instantiated by constants when the routines are executed. The associated operators will also have parameters, but as we shall soon see, these can be left free at the time they are applied to a model.

In order to chain together a sequence of operators to achieve a given goal, the problem solver must have descriptions of the operators.

The descriptions used by STRIPS consist of three major components:

- (1) Name of the operator and its parameters,
- (2) Preconditions, and
- (3) Effects.

The first component consists merely of the name of the operator and the parameters taken by the operator. The second component is a formula in first-order logic. The operator is applicable in any world model in which the precondition formula is a theorem. For example, the operator $\text{push}(u,x,y)$ which models the action of the robot pushing an object u from location x to location y might have as a precondition formula

$$(\exists x,u)[AT(u,x) \wedge ATR(x)] .$$

The third component of an operator description defines the effects (on a set of wffs) of applying the operator. We shall discuss the process of computing effects in some detail since it plays a key role in STRIPS. When an operator is applied, certain wffs in the world model are no longer true (or at least we cannot be sure that they are true) and certain other wffs become true. Thus to compute one world model from another involves copying* the world model and in this copy deleting some of the wffs and

* In our implementation of STRIPS we employ various bookkeeping techniques to avoid copying; these will be described in a later section.

adding others. Let us deal first with the set of wffs that should be added as a result of an operator application.

The set of wffs to be added to a world model depends on the results of the routine modeled by the operator. These results are not completely specified until all of the parameters of the routine are instantiated by constants. For example, the operator $\text{goto}(x,y)$ might model the robot moving from location x to location y for any two locations x and y . When this operator's routine is executed, the parameters x and y must be instantiated by constants. However, we have designed STRIPS so that an operator can be applied to a world model with any or all of the operator's parameters left uninstantiated. For example, suppose we apply the operator $\text{goto}(a,x)$ to a world model in which the robot is at some location^{*} a . If the parameter x is unspecified, so will be the resulting world model. We could say that the application of $\text{goto}(a,x)$ creates a family or schema of world models parameterized by x . The power and efficiency of STRIPS is increased by searching in this space of world model families rather than in the larger space of individual world models.

If we are to gain this reduction in search space size, then we must be able to describe with a single set of predicate calculus wffs the world model family resulting from the application of an operator with free parameters. One way in which this can be done is to use a state term in each literal of each wff. Thus, the principal effect of applying the operator $\text{goto}(a,x)$ to some world model s_0 , say, is to add the wff

$$(\forall x) (\exists s) \text{ATR}(x,s)$$

* We shall adopt the convention of using letters near the beginning of the alphabet ($a,b,c,$ etc.) to stand for constants and letters near the end of the alphabet ($u,v,w,x,$ etc.) as variables.

which states that for all values of the parameter x , there exists a world model s in which the robot is at x . With expressions of this sort, a set of wffs can represent families of world models. Selecting specific values for the parameters selects specific members of the family.

Anticipating the use of a resolution-based theorem prover in STRIPS, we shall always express formulas in clause form.¹

Then the formula above would be written

$$\text{ATR}(x, \text{goto}'(a, x, s_0))$$

where $\text{goto}'(a, x, s_0)$ is a function of x replacing the existentially quantified state variable. The value of $\text{goto}'(a, x, s_0)$, for any x , is that world model produced by applying the operator $\text{goto}(a, x)$ to world model s_0 . Recall that any variables (such as x in the formula above) occurring in a clause have implicit universal quantification.

The description of each operator used in STRIPS contains a list of those clauses to be added when computing a new world model. This list is called the add list.

The description of an operator also includes information about which clauses can no longer be guaranteed true and must therefore be deleted in constructing a new world model. For example, if the operator $\text{goto}(a, y)$ is applied, we must delete any clause containing the atom^{*} $\text{ATR}(a)$. Each operator description contains a list of atoms, called the delete list, that is used to compute which clauses should be deleted. Our rule for creating a new world model is to delete any clauses containing atoms (negated or unnegated) that are instances of atoms on the delete list. We also delete any clauses containing atoms of which the atoms on

* An atom is a single predicate letter and its arguments.

on the delete list are instances. The application of these rules might sometimes delete some clauses unnecessarily, but we want to be guaranteed that the new world model will be consistent if the old one was.

When an operator description is written, it may not be possible to name explicitly all the atoms that should appear on the delete list. For example, it may be the case that a world model contains clauses that are derived from other clauses in the model. Thus from $AT(B1,a)$ and from $AT(B2,a+\Delta)$ we might derive $NEXTTO(B1,B2)$ and insert it into the model. Now, if one of the clauses on which the derived clause depends is deleted, then the derived clause must be deleted also.

We deal with this problem by defining a set of primitive predicates (e.g., AT , ATR , BOX) and relating all other predicates to this primitive set. In particular, we require the delete list of an operator description to indicate all the atoms containing primitive predicates which should be deleted when the operator is applied. Also, we require that any non-primitive clause in the world model have associated with it those primitive clauses on which its validity depends. (A primitive clause is one which contains only primitive predicates.) For example, the clause $NEXTTO(B1,B2)$ would have associated with it the clauses $AT(B1,a)$ and $AT(B2,a+\Delta)$.

By using these conventions we can be assured that primitive clauses will be correctly deleted during operator applications, and that the validity of nonprimitive clauses can be determined whenever they are to be used in a deduction by checking to see if all of the primitive clauses on which the nonprimitive clause depends are still in the world model.

In the next section, we shall describe the search process for STRIPS and also present a specific example in which the process of operator application is examined in detail.

III THE OPERATION OF STRIPS

A. Computing Differences and Relevant Operators

In a very simple problem-solving system we might first apply all of the applicable operators to the initial world model to create a set of successor models. We would continue to apply all applicable operators to these successors and to their descendants until a model was produced in which the goal formula was a theorem. Checking to see which operators are applicable and to see if the goal formula is a theorem are theorem-proving tasks that could be accomplished by a deductive system such as QA3. However, since we envision uses in which the number of operators applicable to any given world model might be quite large, such a simple system would generate an undesirably large tree of world models and would thus be impractical.

Instead we would like to use the GPS strategy of extracting "differences" between the present world model and the goal and of identifying operators that are "relevant" to reducing these differences. Once a relevant operator has been determined, we attempt to solve the subproblem of producing a world model to which it is applicable. If such a model is found then we apply the relevant operator and reconsider the original goal in the resulting model.

When an operator is found to be relevant, it is not known where it will occur in the completed plan; that is, it may be applicable

to the initial model and therefore be the first operator applied, its effects may imply the goal so that it is the last operator applied, or it may be some intermediate step toward the goal. Because of this flexibility, the STRIPS search strategy combines many of the advantages of both forward search (from the initial model toward the goal) and backward search (from the goal toward the initial model).

Two key steps in this strategy involve computing differences and finding operators relevant to reducing these differences. One of the novel features of our system is that it uses a theorem prover as an aid in these steps. The following description of these processes assumes that the reader is familiar with the terminology of resolution-based theorem-proving systems.

Suppose we have a world model consisting of a set, S , of clauses, and that we have a goal formula whose negation is represented by the set, G , of clauses. The difference-computing mechanism attempts to find a contradiction for the set $S \cup G$ using a resolution theorem prover such as QA3. (The theorem prover would likely use, at least, the set-of-support strategy with G the set receiving support.) If a contradiction is found, then the "difference" is nil and STRIPS would conclude that the goal is satisfied in S .

Our interest at the moment though is in the case in which QA3 cannot find a contradiction after investing some prespecified amount of effort. Let R be the set consisting of the clauses in G and the resolvents produced by QA3 which are descendants of G . Any set of clauses D in

R can be taken as a "difference" between S and the goal in the sense that if a world model were found in which a clause in D could be contradicted, then it is likely* that the proof of the goal could be completed in that model.

STRIPS creates differences by heuristically selecting subsets of R, each of which acts as a difference. The selection process considers such factors as the number of literals in a clause, at what level in the proof tree a clause was generated, and whether or not a clause has any descendants in the proof tree.

The quest for relevant operators proceeds in two steps. In the first step an ordered list of candidate operators is created for each difference set. The selection of operators for this list is based on a simple comparison of the clauses in the difference set with the add lists in the operator descriptions. For example, if a difference set contained a clause having in it the robot position predicate ATR, then the operator goto would be considered a candidate operator for that difference.

The second step in finding an operator relevant to a given difference set involves employing QA3 to determine if clauses on the add list of a candidate operator can be used to "resolve away" (i.e., continue the proof of) any of the clauses in the difference set. If, in fact, QA3 can produce new resolvents which are descendants of the add list clauses, then the candidate operator (properly instantiated) is considered to be a relevant operator for the difference set.

* That is, a proof could be completed if this new model still allows a deduction of this clause in D.

To complete the operator-relevance test STRIPS must determine which instances of the operator are relevant. For example, if the difference set consists of the unit clauses $\text{-ATR}(a)$ and $\text{-ATR}(b)$, then $\text{goto}(x,y)$ is a relevant operator only when y is instantiated by a or b . Each new resolvent which is a descendant of the operator's add list clauses is used to form a relevant instance of the operator by applying to the operator's parameters the same instantiations that were made during the production of the resolvent. Hence the consideration of one candidate operator may produce several relevant operator instances.

One of the important effects of the difference-reduction process is that it usually produces specific instances for the operator parameters. Furthermore, these instances are likely to be those occurring in the final solution, thus helping to narrow the search process. So, although STRIPS has the ability to consider operators with uninstantiated parameters, it also has a strong tendency toward instantiating these parameters with what it considers to be the most relevant constants.

B. The STRIPS Executive

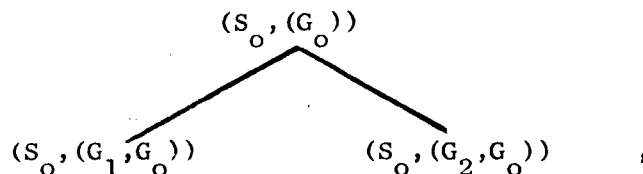
STRIPS begins by attempting to form differences between the initial world model, s_0 , and the main goal (as described in the previous section). If no differences are found, then the problem is trivially solved. If differences are found, then STRIPS computes a set of operators relevant to reducing those differences.

Suppose, for example, that STRIPS finds two instantiated operators, OP_1 and OP_2 , relevant to reducing the differences between s_0 and the main goal. Let the (instantiated) precondition formulas for

these operators be denoted by PC_1 and PC_2 , respectively. Thus STRIPS has found two ways to work on the main problem:

- (1) Produce a world model to which OP_1 is applicable, apply OP_1 , and then produce a world model in which the main goal is satisfied, or
- (2) Produce a world model to which OP_2 is applicable, apply OP_2 , and then produce a world model in which the main goal is satisfied.

STRIPS represents such solution alternatives as nodes on a search tree. The tree for our example can be represented as follows:



where G_0, G_1 , and G_2 are sets of clauses corresponding to the negations of the main theorem, PC_1 and PC_2 , respectively.

In general, each node of the search tree has the form $\langle\langle$ world model $\rangle\rangle, \langle$ goal list \rangle . The subgoal being considered for solution at each node is the first goal on that node's goal list. The last goal on each list is the negation of the main goal, and each subgoal is the negation of the preconditions of an operator. Hence, each subgoal in a goal list represents an attempt to apply an operator which is relevant to achieving the next goal in the goal list.

Whenever a new node, $(s_i, (G_m, G_{m-1}, \dots, G_1, G_0))$, is constructed and added to the search tree as a descendant of some existing node, the new node is tested for goal satisfaction. This test is performed by QA3 which looks for a contradiction to $s_i \cup G_m$.

If a contradiction is found and m is 0 (i.e., the node has the form $(s_i, (G_0))$), then the main goal is satisfied in s_i and the problem is solved. If a contradiction is found and m is not 0, then G_m is the negation of a precondition formula for an operator that is applicable in s_i . STRIPS produces a new world model, s'_i , by applying to s_i the operator corresponding to G_m . The node is changed to $(s'_i, (G_{m-1}, \dots, G_1, G_0))$ and the test for goal satisfaction is performed on it again. This process of changing the node continues until a goal is encountered which is not satisfied or until the problem is solved.

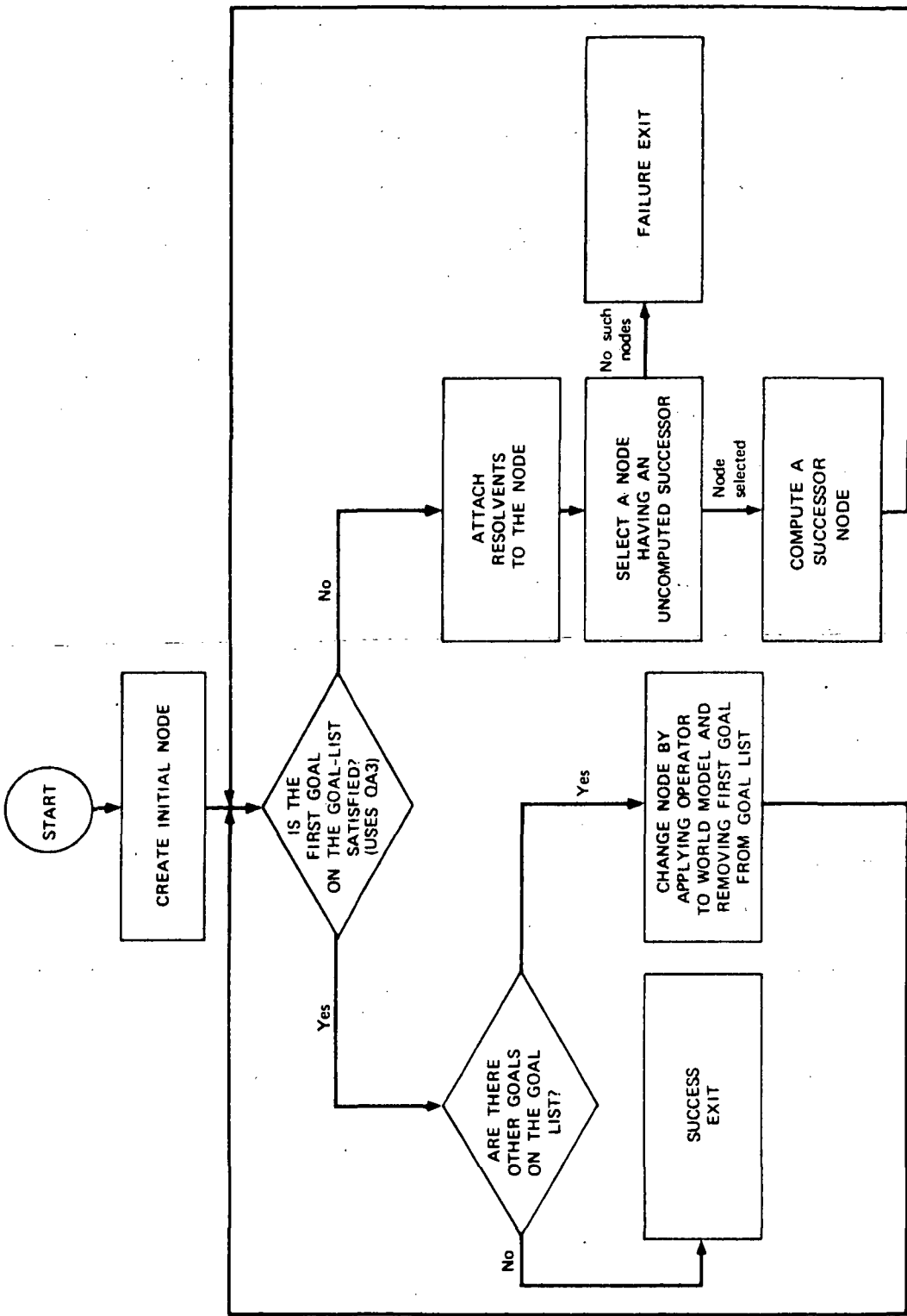
If no contradiction is found in the goal satisfaction test, QA3 will return a set R of clauses consisting of the clauses in G_m and resolvents that are descendants of clauses in G_m . This set of resolvents is attached to the node and is used for generating successors to the node.

The process for generating the successors of a node $(s_i, (G_m, G_{m-1}, \dots, G_1, G_0))$ with R attached involves forming difference sets $\{D_i\}$ from R and finding operator instances relevant to reducing these differences (as described in the previous section). For each operator instance found to be relevant, a new offspring node is created. This new node is formed with the same world model and goal list as its parent node. The goal of finding a world model in which the relevant operator instance can be applied is added to the new node. This is done by creating the appropriate instance of the operator's preconditions and adding the negation of the instantiated preconditions to the beginning of the new node's goal list.

Since the number of operators relevant to reducing sets of differences might be rather large in some cases, it is possible that a given node in the search tree might have a large number of successors. Even before the successors are generated, though, we can order them according to the heuristic merit of the operators and difference sets used to generate them. The process of computing a successor node can be rather lengthy, and for this reason STRIPS actually computes only that single next successor judged to be best. STRIPS adds this successor node to the search tree, performs a goal-satisfaction test on it, and then selects another node from the set of nodes which still have uncomputed successors. STRIPS must therefore associate with each node the sets of differences and candidate operators it has already used in creating successors.

STRIPS will have a heuristic mechanism to select nodes with uncomputed successors to work on next. For this purpose we will use an evaluation function that takes into account such factors as the number and types of literals in the remaining goal formulas, the number of remaining goals, and the number and types of literals in the difference sets.

A simple flowchart of the STRIPS executive is shown in Figure 1.



TA-8259-30

FIGURE 1 FLOWCHART FOR THE STRIPS EXECUTIVE

C. An Example

An understanding of how STRIPS works is aided by tracing through a simple example. Consider the configuration shown in Figure 2 consisting of two objects B and C and a robot R at places b, c, and a, respectively. The problem given to STRIPS is to achieve a configuration in which object B is at place k and in which object C is not at place c.

The existentially quantified theorem representing this problem can be written

$$(\exists s) [AT(B,k,s) \wedge \sim AT(C,c,s)] \quad .$$

If we can find an instance of s (in terms of a composition of operator applications) that satisfies this theorem, then we will have solved the problem. The negation of the theorem is

$$G_0 : \sim AT(B,k,s) \vee AT(C,c,s) \quad .$$

Let us suppose that STRIPS is to compose a solution using the two operators goto and push. These operators can be described as follows:

1. push(u,x,y): Robot pushes object u from place x to place y.

Precondition formula:

$$(\exists u,x,s) [AT(u,x,s) \wedge ATR(x,s)]$$

Negated precondition formula:

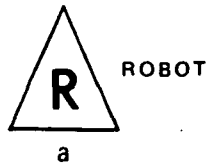
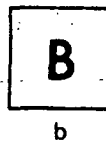
$$\sim AT(u,x,s) \vee \sim ATR(x,s)$$

Delete list:

$$AT(u,x,s)$$

$$ATR(x,s)$$

k



TA-8259-31

FIGURE 2 CONFIGURATION OF OBJECTS AND ROBOT FOR EXAMPLE PROBLEM

Add list:

$$AT(u,y,push'(u,x,y,s^*))$$
$$ATR(y,push'(u,x,y,s^*))$$

where s^* is the state to which the operator is applied.

2. goto(x,y): Robot goes from place x to place y.

Precondition formula:

$$(\exists x,s)ATR(x,s)$$

Negated precondition formula:

$$\sim ATR(x,s)$$

Delete list:

$$ATR(x,s)$$

Add list:

$$ATR(y,goto'(x,y,s^*))$$

The initial configuration can be described by the following world model:

$$s_0 : ATR(a,s_0)$$
$$AT(B,b,s_0)$$
$$AT(C,c,s_0) \quad .$$

In addition, we have a universal formula, true in all world models, that states if an object is in one place, then it is not in a different place:

$$F: (\forall u,x,y,s)[AT(u,x,s) \wedge (x \neq y) \Rightarrow \sim AT(u,y,s)] \quad .$$

The clause form of this formula is

$$F': \sim AT(u,x,s) \vee (x = y) \vee \sim AT(u,y,s) \quad .$$

We assume that F' is adjoined to all world models.

STRIPS first constructs the node N_0 , consisting of the list $(s_0, (G_0))$, as the root of the problem-solving tree and tests it for a solution by attempting to find a contradiction for the set $s_0 \cup \{G_0\}$. No contradiction is found but some resolvents can be obtained; among them are two resolvents of G_0 and F' :

$$R_1: \sim AT(B, k, s) \vee (c = y) \vee \sim AT(C, y, s)$$

and
$$R_2: \sim AT(B, k, s) \vee (x = c) \vee \sim AT(C, x, s)$$

Additional resolvents can be produced also, but these happen all to be tautologies and can thus be eliminated.[†] A sophisticated system would detect that R_1 and R_2 are identical, so let us suppose that R_1 is the only resolvent attached to N_0 .

Next STRIPS selects a node (N_0 is now the only one available) and begins to generate successors. First it selects a difference set D_1 from the set of resolvents attached to N_0 . In this case it sets $D_1 = \{R_1\}$. Then STRIPS composes a list L of candidate operators for reducing D_1 . Here L would consist of the single element push.

Next STRIPS attempts to reduce D_1 using clauses on the add list of push. Again using theorem-proving methods we obtain two resolvents from D_1 and $AT(u, y, \text{push}'(u, x, y, s^*))$:

$$\sim AT(B, k, \text{push}'(C, x, y, s^*)) \vee (c = y)$$

and
$$\sim AT(C, y, \text{push}'(B, x, k, s^*)) \vee (c = y)$$

[†]We are assuming a set-of-support strategy with the initial support set consisting only of the negated theorem.

Assuming that these resolutions represent acceptable reductions in the difference, we extract the state terms of the resolvents to yield appropriate instances of the relevant operator. This gives us:

$$OP_1: \text{push}(C, x, y)$$

and $OP_2: \text{push}(B, x, k)$.

Next, we construct the negated versions of the precondition formulas for OP_1 and OP_2 :

$$G_1: \sim AT(C, x, s) \vee \sim ATR(x, s)$$

and $G_2: \sim AT(B, x, s) \vee \sim ATR(x, s)$.

These formulas are then used to construct two successor nodes

$$N_1: (s_0, (G_1, G_0))$$

and $N_2: (s_0, (G_2, G_0))$.

These nodes would be immediately tested for solutions. For brevity, let us consider just N_1 . In testing for a solution STRIPS attempts to find a contradiction for $s_0 \cup G_1$.

Again no contradiction is found, but the following resolvents are obtained:

$$R_3: \sim ATR(c, s_0) \text{ from } G_1 \text{ and } AT(C, c, s_0)$$

and $R_4: \sim AT(C, a, s_0) \text{ from } G_1 \text{ and } ATR(a, s_0)$.

Although these clauses represent differences between s_0 and G_1 , we do not insist that these differences be reduced in s_0 . We would accept a reduction occurring in any world model, so STRIPS rewrites the clauses as:

$$R_3': \sim ATR(c, s)$$

and $R_4': \sim AT(C, a, s)$.

These clauses refer to preconditions for pushing object C. To contradict R_3' the robot must be at c; to contradict R_4' object C must be at a. Suppose our system recognizes that an attempt to contradict R_4' is circular and attaches just the set $\{R_3'\}$ to node N_1 .

Next STRIPS selects a node for consideration. Suppose it selects N_1 . In generating successors, it sets the difference set, D_2 , to $\{R_3'\}$.

The list of operators useful for reducing D_2 consists only of goto. STRIPS now attempts to perform resolutions between the clauses on the add list of goto and D_2 . The clause in D_2 resolves with $ATR(y, goto'(x, y, s^*))$ to yield nil, and answer extraction produces the instance substituted for the state term, namely

$$s = goto'(x, c, s^*)$$

Thus STRIPS identifies the following instance of goto:

$$OP_3: goto(x, c)$$

The associated negated precondition is

$$G_3: \sim AT(R, x, s)$$

STRIPS then constructs the successor node

$$N_3: (s_0, (G_3, G_1, G_0))$$

and immediately attempts to find a contradiction for $s_0 \cup G_3$. Here a contradiction is obtained, and answer extraction yields the state term:

$$goto'(a, c, s_0)$$

Thus STRIPS applies goto(a, c) to s_0 to yield

$$s_1: ATR(c, goto'(a, c, s_0)) \\ AT(B, b, goto'(a, c, s_0)) \\ AT(C, c, goto'(a, c, s_0))$$

Node N_3 is then changed to

$$N_4: (s_1, (G_1, G_0))$$

and STRIPS immediately checks for a contradiction for $s_1 \cup G_1$. Again a contradiction is found; answer extraction produces the following instances for x and s :

$$x = c$$

and $s = \text{goto}'(a, c, s_0)$.

Thus STRIPS applies the following instance of OP_1 :

$$\text{push}(C, c, y)$$
 .

The result is the world model family s_2 consisting of the following clauses:

$$s_2: \text{ATR}(y, \text{push}'(C, c, y, \text{goto}'(a, c, s_0)))$$

$$\text{AT}(B, b, \text{push}'(C, c, y, \text{goto}'(a, c, s_0)))$$

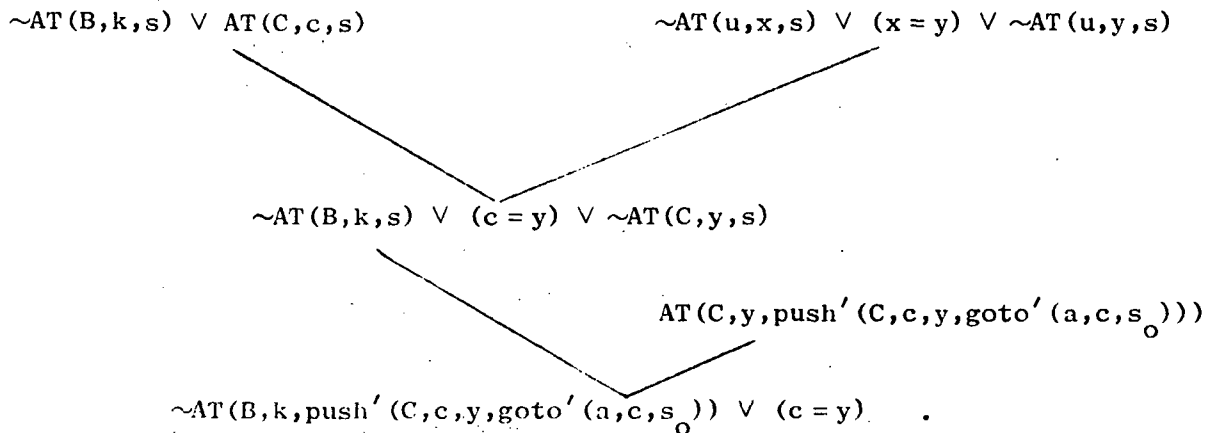
$$\text{AT}(C, y, \text{push}'(C, c, y, \text{goto}'(a, c, s_0)))$$
 .

Note that this application of the operator push involved an uninstan-
tiated parameter, y .

Node N_4 is then changed to

$$N_5: (s_2, (G_0))$$

and STRIPS checks for a contradiction for $s_2 \cup G_0$. In doing so it produces the following tree of resolutions:



The clause at the root produces one of the resolvents to be attached to N_5 , namely

$$R_5: \sim AT(B,k,s) \vee (c=y) \quad .$$

Suppose STRIPS selects N_5 next and begins generating successors based on a difference $D_3 = \{R_5\}$. The operator list for this difference consists solely of push, and the relevant instance of push is found to be

$$OP_4: \text{push}(B,x,k) \quad .$$

Its (negated) precondition is

$$G_4: \sim AT(B,x,s) \vee \sim ATR(x,s) \quad .$$

A successor node to N_5 is then

$$N_6: (s_2, (G_4, G_0)) \quad .$$

STRIPS then finds a contradiction between s_2 and G_4 and extracts

$$s = \text{push}'(C,c,b,\text{goto}'(a,c,s_0))$$

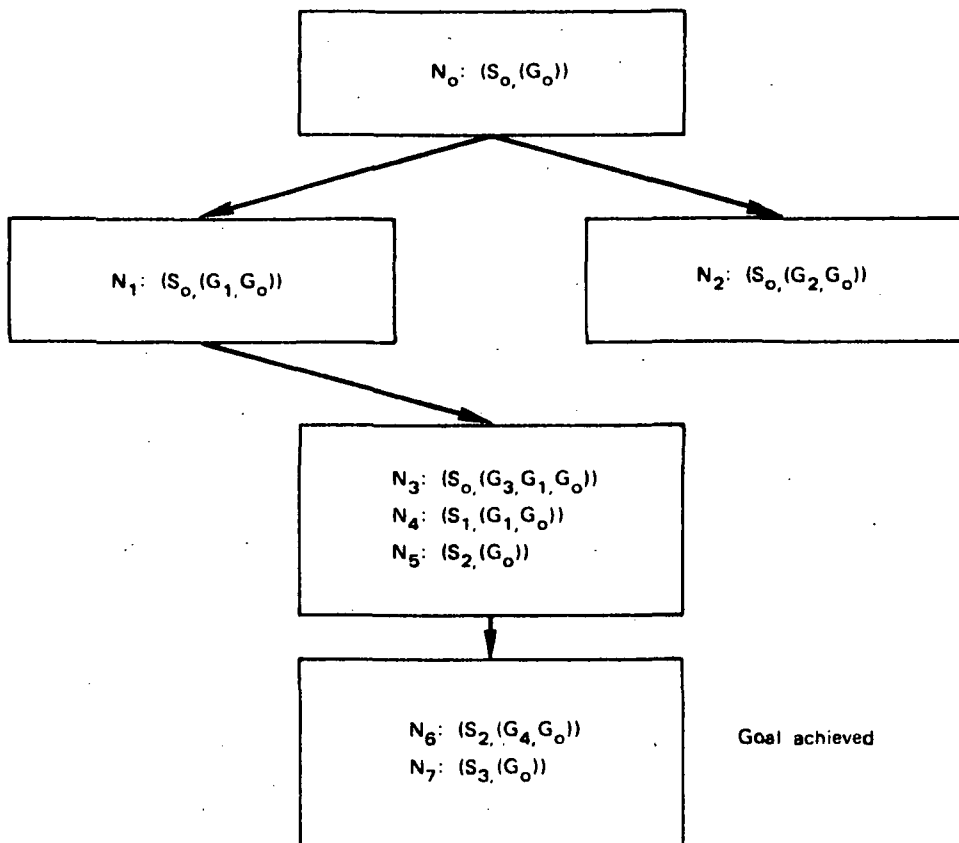
and $x = b$. Therefore, it applies $\text{push}(B,b,k)$ to an instance of s_2 (with $y = b$) to yield

$$\begin{aligned} s_3: & \text{ATR}(k,\text{push}'(B,b,k,\text{push}'(C,c,b,\text{goto}'(a,c,s_0)))) \\ & \text{AT}(B,k,\text{push}'(B,b,k,\text{push}'(C,c,b,\text{goto}'(a,c,s_0)))) \\ & \text{AT}(C,b,\text{push}'(B,b,k,\text{push}'(C,c,b,\text{goto}'(a,c,s_0)))) \quad . \end{aligned}$$

Node N_6 is then changed to node

$$N_7: (s_3, (G_0)) \quad .$$

STRIPS can find a contradiction between s_3 and G_0 [assuming that the equality predicate $(b=c)$ can be evaluated to be false] and exits successfully. The successful plan is embodied in the state term for s_3 . We show the solution path in the STRIPS problem-solving tree in Figure 3.



TA-8259-32

FIGURE 3 SEARCH TREE FOR EXAMPLE PROBLEM

D. Efficient Representation of World Models

A primary design issue in the implementation of a system such as STRIPS is how to satisfy the storage requirements of a search tree in which each node may contain a different world model. We would like to use STRIPS in a robot or question-answering environment where the initial world model may consist of hundreds of wffs. For such applications it is infeasible to recopy completely a world model each time a new model is produced by application of an operator.

We have dealt with this problem in STRIPS by first making the assumption that most of the wffs in a problem's initial world model will not be changed by the application of operators. This is certainly true for the class of robot problems we are currently concerned with. For these problems most of the wffs in a model describe rooms, walls, doors, and objects, or specify general properties of the world which are true in all models. The only wffs that might be changed in this robot environment are the ones that describe the status of the robot and any objects which it manipulates.

Given this assumption, we have implemented the following scheme for handling multiple world models. All the wffs for all world models are stored in a common memory structure. Associated with each wff (i.e., clause) is a visibility flag, and QA3 has been modified to consider only clauses from the memory structure which are marked visible. Hence, we can "define" a particular world model for QA3 by marking that model's clauses visible and all other clauses invisible. When clauses are entered into the initial world model they are marked visible and

given a variable as a state term. Clauses not changed will remain visible throughout STRIPS' search for a solution.

Each world model produced by STRIPS is defined by two clause lists. The first list, DELETIONS, names all those clauses from the initial world model which are no longer present in the model being defined. The second list, ADDITIONS, names all those clauses in the model being defined which are not also in the initial model. These lists represent the changes in the initial model needed to form the model being defined, and our assumption implies they will contain only a small number of clauses.

To specify a given world model to QA3, STRIPS marks visible the clauses on the model's ADDITIONS list and marks invisible the clauses on the model's DELETIONS list. When the call to QA3 is completed, the visibility markings of these clauses are returned to their previous settings.

When an operator is applied to a world model, the DELETIONS list of the new world model is a copy of the DELETIONS list of the old model plus any clauses from the initial model which are deleted by the operator. The ADDITIONS list of the new model consists of the clauses from the old model's ADDITIONS list as transformed by the operator plus the clauses from the operator's add list.

To illustrate this implementation design we list below the way in which the world models described in the example of the previous section are represented:

s_0 : ATR(a,s)
 AT(B,b,s)
 AT(C,c,s)

s_1 : DELETIONS: ATR(a,s)
 ADDITIONS: ATR(c,goto'(a,c,s₀))

s_2 : DELETIONS: ATR(a,s)
 AT(C,c,s)
 ADDITIONS: ATR(y,push'(C,c,y,goto'(a,c,s₀)))
 AT(C,y,push'(C,c,y,goto'(a,c,s₀)))

s_3 : DELETIONS: ATR(a,s)
 AT(C,c,s)
 AT(B,b,s)
 ADDITIONS: ATR(k,push'(B,b,k,push'(C,c,b,goto'(a,c,s₀))))
 AT(B,k,push'(B,b,k,push'(C,c,b,goto'(a,c,s₀))))
 AT(C,c,push'(B,b,k,push'(C,c,b,goto'(a,c,s₀))))

IV FUTURE PLANS AND PROBLEMS

The implementation of STRIPS now being completed can be extended in several directions. These extensions will be the subject of much of our problem-solving research activities in the immediate future. We shall conclude this note by briefly mentioning some of these.

We have seen that STRIPS constructs a problem-solving tree whose nodes represent subproblems. In a problem-solving process of this sort, there must be a mechanism to decide which subproblem to work on next. We have already mentioned some of the factors that might be incorporated in an evaluation function by which subproblems can be ordered according to heuristic merit. We expect to devote a good deal of effort to devising and experimenting with various evaluation functions and other ordering techniques.

Another area for future research concerns synthesis of more complex procedures than those consisting of simple linear sequences of operators. Specifically we want to be able to generate procedures involving iteration (or recursion) and conditional branching. In short, we would like STRIPS to be able to generate computer programs. Several researchers^{4,8,9} have already considered the problem of automatic program synthesis and we expect to be able to use some of their ideas in STRIPS.

Our implementation of STRIPS is designed to facilitate the definition of new operators by the user. Thus the problem-solving power of STRIPS can gradually increase as its store of operators grows.

An idea that may prove useful in robot applications concerns defining and using operators to which there correspond no execution routines. That is, STRIPS may be allowed to generate a plan containing one or more operators that are fictitious. This technique essentially permits STRIPS to assume that certain subproblems have solutions without actually knowing how these solutions are to be achieved in terms of existing robot routines. When the robot system attempts to execute a fictitious operator, the subproblem it represents must first be solved (perhaps by STRIPS). (In human problem solving, this strategy is employed when we say: "I won't worry about that [sub] problem until I get to it.")

We are also interested in getting STRIPS to define new operators for itself based on previous problem solutions. One reasonable possibility is that after a problem represented by (S_0, G_0) is solved, STRIPS could automatically generate a fictitious operator to represent the solution. It would be important to try to generalize any constants

appearing in G_0 ; these would then be represented by parameters in the fictitious operator. The structure of the actual solution would also have to be examined in order to extract a precondition formula, delete list, and add list for the fictitious operator.

A more ambitious undertaking would be an attempt to synthesize automatically a robot execution routine corresponding to the new operator. Of course, this routine would be composed from a sequence of the existing routines corresponding to the individual existing operators used in the problem solution. The major difficulty concerns generalizing constants to parameters so that the new routine is general enough to merit saving. Hewitt¹⁰ discusses a related problem that he calls "procedural abstraction." He suggests that from a few instances of a procedure, a general version can sometimes be synthesized. We expect that our generalization problem will be aided by an analysis of the structure of the preconditions and effects of the individual operators used in the problem solution.

ACKNOWLEDGMENT

The development of the ideas embodied in STRIPS has been the result of the combined efforts of the present authors, Bertram Raphael, Thomas Garvey, John Munson, and Richard Waldinger, all members of the Artificial Intelligence Group at SRI.

REFERENCES

1. N. J. Nilsson, Problem-Solving Methods in Artificial Intelligence (McGraw-Hill Book Company, New York, to appear in April 1971).
2. C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems," in Machine Intelligence 4, B. Meltzer and D. Michie (Eds.), pp. 183-205 (American Elsevier Publishing Co., Inc., New York, 1969).
3. G. Ernst and A. Newell, GPS: A Case Study in Generality and Problem Solving, ACM Monograph Series (Academic Press, 1969).
4. C. Green, "Application of Theorem Proving to Problem Solving," Proc. Intl. Joint Conf. on Artificial Intelligence, Washington, D.C. (May 1969).
5. D. Luckham and N. Nilsson, "Extracting Information from Resolution Proof Trees," Artificial Intelligence (to appear).
6. J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Machine Intelligence 4, B. Meltzer and D. Michie (Eds.), pp. 463-502 (American Elsevier Publishing Co., Inc., New York, 1969).
7. B. Raphael, "The Frame Problem in Problem-Solving Systems," Proc. Adv. Study Inst. on Artificial Intelligence and Heuristic Programming, Menaggio, Italy (August 1970).
8. R. Waldinger and R. Lee, "PROW: A Step Toward Automatic Program Writing," Proc. Intl. Joint Conf. on Artificial Intelligence, Washington, D.C. (May 1969).

9. Z. Manna and R. Waldinger, "Towards Automatic Program Synthesis," Artificial Intelligence Group Technical Note 34, Stanford Research Institute, Menlo Park, California (July 1970).
10. C. Hewitt, "Planner: A Language for Manipulating Models and Proving Theorems in a Robot," Artificial Intelligence Memo No. 168 (Revised), Massachusetts Institute of Technology, Project MAC, Cambridge, Massachusetts (August 1970).

APPENDIX E

QA4 WORKING PAPER

October 1970

QA4 WORKING PAPER

by

Johns F. Rulifson
Richard J. Waldinger
Jan Derksen

Artificial Intelligence Group

Technical Note 42

SRI Projects 8721, 8550, 8259

The research reported here is sponsored by National Aeronautics and Space Administration under Contracts NASW-2086 and NAS12-2221 and by Air Force Cambridge Research Laboratories under Contract F19628-70-C-0246.

I GENERAL GOALS OF THE LANGUAGE

A. The Language and Its Data Base

The QA4 language is an enhanced omega-order language^{1*} embedded in a system of control statements. The declarative facets of the language include atomic symbols, tuples, unordered tuples, sets, function definitions, and applications; the imperative facets include (in addition to normal program control features) set iteration, backtracking, and parallelism. The language is intended to be a natural formalism for the description of problem-domain-oriented theorem-proving strategies. Moreover, the specification of problems to be solved by QA4 programs have a natural, compact formulation in the same language. That is, the statement of theorems to be proved or the specification of programs to be written is a task similar in nature to writing theorem provers or program synthesizers. For this reason, the data base for QA4 programs is QA4 expressions. A preliminary description of the QA4 syntax appears in Ref 2.

B. Properties of Expressions

In addition to the syntactic component that uniquely distinguishes it from all other QA4 expressions, every QA4 expression has a property list. This list stores arbitrary properties and their values,

* References are listed at the end of this note.

the values being, in turn, QA4 expressions. The properties are used by QA4 programs both to store information for the interpreter, and to guide strategies and communicate information about the data on which the programs are working. These properties fall into three categories: interpreter bookkeeping, semantic, and pragmatic.

The standard semantic properties of an expression include its value, the set of expressions it is known to equal, the sets of expressions it may not equal. Rules for evaluation and simplification are also semantic properties. It is assumed that partial evaluation or simplification of expressions will be an important strategy in all QA4 problem solvers. The QA4 interpreter comes equipped with such a partial evaluator. It is, however, incomplete, but can be enhanced through the use of appropriate semantic properties. Finally, it is often useful to write a strategy in terms of a particular data structure, say a set. The programs may be clear and concise, making the strategy transparent and flexible. Yet, for reasons of efficiency it may be necessary to represent the set outside the standard QA4 framework, say with a LISP array. Such representation information is handled by the use of semantic properties.

Pragmatic properties are peculiar to each individual problem. The properties are used by strategy programs to communicate and note information about expressions. They take the flavor of statements such as "I've tried this before and it didn't work."

C. Expression Manipulations

Expression manipulation is accomplished by decomposition and construction. Decomposition, in QA4, means naming parts or components of an expression. The naming is done with pattern matching. Patterns may occur at many points in the language: in functional variable bindings, assignment statements, and conditional tests. Transformation of expressions is done through a complete set of constructors: add an element to a set, add onto tuples, or construct a lambda expression, to name a few. There is also a large set of primitive operators on the structural data forms, e.g., set union, arithmetic addition, and Boolean conjunction.

D. Control Statements

In order to solve large problems and carry out long proofs, it is necessary to have highly goal-directed search strategies. Moreover, many of the searches done in QA4 strategy programs simply do not have appropriate numerical means of guiding them. That is, the semantic-pragmatic search techniques are guided by programs making local decisions on current information. Any attempt to centralize the search or have uniform procedures cannot be done easily. For this reason, the QA4 language makes directly available, through statements in the language, many well-known search procedures. This means that each particular problem-domain-oriented strategy program can use appropriate search techniques at its own local level. Strategies may thus search in parallel,

grow search trees, or backtrack whenever such methods are appropriate. Accordingly, one can no longer characterize a QA4 program as doing a particular kind of search while it is problem solving; in most cases, many (if not all) kinds of search are being done.

The search-oriented statements of QA4 fall into three categories:

Iteration over sets--taking the form of selection through patterns and for each statements.

Parallelism--Appearing as coroutines, parallel strategy execution, and when statements.

Backtracking--Taking place in the program failure mechanism and the choice function (choices many times being made from possible matches to a pattern).

II ORGANIZATION OF THE INTERPRETER

A. User Interface

The QA4 programmer views the system as an interactive programming tool. He types commands in the form of QA4 expressions to a top-level function. These commands may input or modify expressions or values of properties of expressions; define, modify, or execute programs; or perform debugging tasks. Roughly speaking, the system is divided into three parts: input/output, editor, and interpreter.

The input/output system is an expression parser, which transforms QA4 infix syntax into prepolish or internal format. The parser uses the BIP package³ and has the advantage of being readily modified.

Similarly, an output function takes the internal expression form and outputs a corresponding infix output stream. Thus, the user always communicates with QA4 in an infix mathematical-style notation.

The editor is still conceptual. While we feel it is an essential part of a useful human-oriented system, it is yet to be specified.

The QA4 interpreter is an EVAL function resembling LISP EVAL. It accepts QA4 expressions and, with the aid of an extensive library of primitive functions, executes them. At this time we have no plans to make interpretations of expressions that do not have an immediate, obvious value (say, FORALL statements). We hope that experience with theorem-proving programs will show ways of automatically extending the basic EVAL.

B. Expression Storage

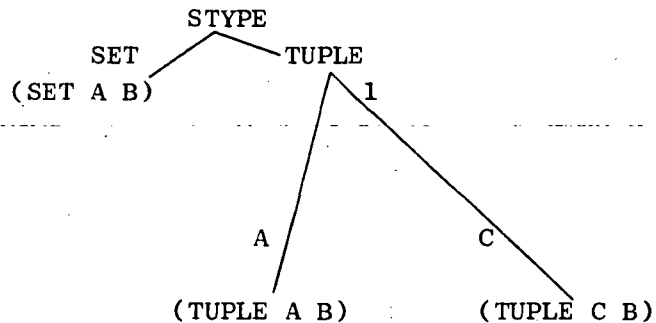
The storage and retrieval of expressions is fundamental to the QA4 system. That is, given a syntactic form for an expression, a fundamental operation is to look the form up and find the properties already assigned or known about the form. This is an extension of LISP's atom property feature to expressions in general. Internally, a QA4 expression is a property list consisting of a property EXPV, whose value contains the syntactic information about the expression, and whose remaining properties are semantic or pragmatic. When an expression is stored, a lookup is made to determine whether or not the expression has been stored

before. If so, the old expression is returned, and if not, a new expression is added to the general store. Thus, only one copy of each expression is retained by the system.

The storage mechanism is a discrimination net. To understand the workings of the net, suppose the system contained only the expressions, in internal format,

(SET A B), (TUPLE A B), (TUPLE C B) .

The net automatically created for storing these expressions might be



The net is a tree. Each node of the tree contains

- (1) A function, which extracts an atomic piece of syntactic information, and
- (2) Either a terminal node or a list of branches. (A terminal node contains an expression, and a branch is a pair--an atom and another node).

A syntactic form is looked up in the net by applying the feature extraction at the top node, choosing the appropriate branch, and continuing until a terminal node is reached or there is no appropriate

branch. If no branch exists, then the expression does not occur in the net and a new terminal node may be added.

When a terminal node is reached, the input expression must be checked against the syntactic property on the expression at the terminal node. If they match, all is well and the property list for the form has been found. If they do not match, a new branching node must be created. To construct the feature selector the two expressions are compared in a structural depth-first manner until the first difference is noted. The results of this search are encoded into a list and installed as the feature selector of the new node. A terminal node for the new expression is constructed, the two new branches made up, and the net is transformed to hold the property list for the new form.

If two QA4 expressions are identical except for the names of their bound variables, they go into the same internal representation. Thus, bound variables may not be used as selector functions. Moreover, in order to store sets and bags in the net, an index is assigned to each element of a set or bag expression the first time it is stored. If the same set is then stored a second time (perhaps with some expressions permuted), the elements are first sorted by the index numbers and then discriminated upon syntactically. Thus, if a user types in the set {A,B,C}, the elements are assigned indices $A \leftarrow 1$, $B \leftarrow 2$, $C \leftarrow 3$. If the set {C,B,A} is entered, it is sorted into {A,B,C} and then found to

already occur. The net functions also maintain statistics concerning the number of references made to each expression and discrimination for future optimizations.

C. Equality Partitions

The efficient treatment of the equality predicate is crucial to the operation of any problem-solving system. Rather than axiomatize the equality rules, we have built them into the QA4 system by introducing equality partitions. Each expression in a context has (as its value property for that context name) the set of expressions known to be logically equal to it in that context. When two expressions are asserted or proved equal in a context, their "equality sets" are merged to form a new set for each. Moreover, each expression has (in context) a set of sets of expressions that are known to be unequal to the given expression. That is, each set in the "unequal set" contains a set of expressions known to be not all equal. Again, when a new equality assertion is made, these sets are updated correspondingly. Consequently, whenever an equality assertion causes a contradiction via the equality rules, it is immediately known. An additional advantage to maintaining the equality information is to be able to select the "best" expression equal to a given expression for a certain purpose.

III CONTEXTS

A. Intent and Uses

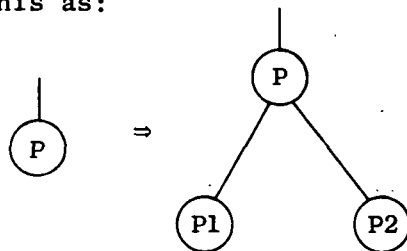
Variable bindings are implemented in the QA4 interpreter with a "context" mechanism. This method of storing all the changeable property values of expressions simplifies the execution of parallelism and backtracking in the interpreter. The same facilities, moreover, are made available to the users as a method of data manipulation in programs dealing with the frame-problem, conditional proofs, or variable bindings. The mechanism simulates a branching pushdown stack. Each node in the tree corresponds to a process or state of the world. When a process changes properties of an expression, the changes are only effective for the process and its descendants. The property values of the ancestors of the process are unchanged.

B. Example

1. Coroutines

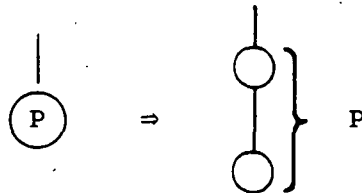
For example, suppose a process P is being interpreted, and it creates two coroutine subprocesses P1 and P2. With each creation, the interpreter creates a new context, and each is an extension of P.

We might represent this as:

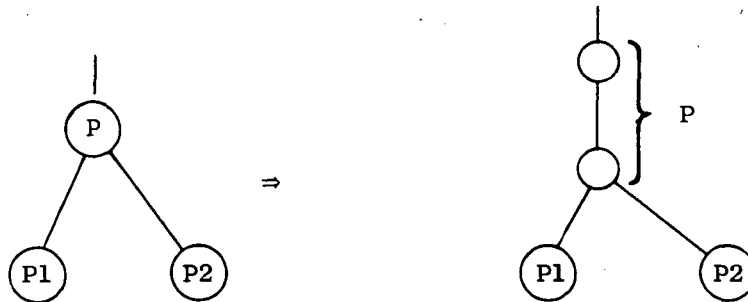


2. Backtracking

Backtracking is slightly different. If P is terminal (that is, it has no subprocesses when a backtracking point is reached), then a new context is created; however, the new context is an extension of P. This is done so that further changes in variable values in P will not destroy the old values, and the state at the backtracking point can be readily restored:



If P already had subprocesses, then the new context is an extension of P, which interposes itself between the original P and the subprocesses:



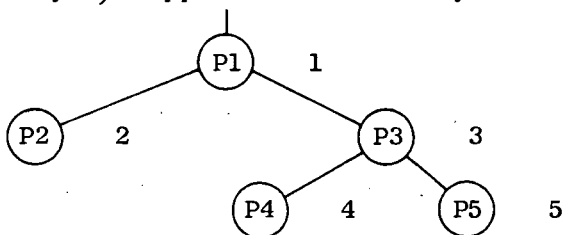
C. A Note of Caution

When the interpreter and programs use the same data base, care must be taken by user programs during property list manipulation. These concerns come naturally to a LISP programmer who confronts the same problem when he uses properties of atoms. The usefulness of the feature, however, certainly makes it worthwhile. The problems of the interpreter and user programs are very similar, and mechanisms useful

for one are probably useful for the other. It is important, therefore, that QA4 programmers fully understand the context mechanism and exploit it in their programs to gain the full power of the language.

D. Implementation

A data item of type context is a list of numbers, say (5, 3, 1). Each number corresponds to a node in the graph representation of the process structure. For example, suppose the current process structure was



then (5, 3, 1), (4, 3, 1), (2, 1), (3, 1) are all possible contexts. Process P3's context is (3, 1), while P4's context is (4, 3, 1). The extension of a context is handled by the function XCTX, which creates a new unique context number and puts it on the front of a context.

The values of properties of expressions are stored as property lists themselves, where the context numbers are property names. For example, an expression might look like:

```
(NETEXPRESSION EXPV (TUPLE 1 2) P1 (CONTEXTLIST 5 Q 3 R)).
```

This internal representation means that the value of property P1 for the tuple <1, 2> was set to Q under a context headed by 5, say (5, 3, 1) and set to R under a context headed by 3, say (3, 1). In the sample above, P3 may have set the value to R, while P5 set it to Q.

E. Lookup

The lookup routine CTXGET takes an expression, a property name, and a context as arguments. If e were a pointer to the above expression, then (CTXGET e "P1" "(3 1)"), would first get the LISP values of property P1, the list (CONTEXTLIST 5 Q 3 R). It would then look for a value under context number 3, and if that fails under 1. In our example, it finds one under 3 and returns R.

F. Changing Contexts

Contexts are popped by the function POPCTX, properties are added with CTXPUT, and removed with CTXREM. The context functions note all current contexts and discard all else during garbage collection.

G. Summary

The whole notion of the discrimination net as a means of accessing expressions is a method of extending the LISP idea of property list from atoms to expressions in general. The inclusion of bound variable expressions and sets in the net causes some concern, but can be handled. The context mechanism is an extension in a similar vein. The values of properties can be with respect to a given state or binding level. LISP programs sometimes do this when the value of a property is treated as a pushdown stack. However, a simple stack is not enough for parallelism and backtracking. The context mechanism appears to be a concise, natural

method of extending the basic notions. It even carries along the features of garbage collection, something which change lists and other approaches have difficulty with.

H. Example

The QA4 theorem prover uses high-level rules of inference. Thus, one QA4 proof step may represent many formal steps. QA4 rules of inference may be very special-purpose: In any situation, we expect the system to select, from a large collection, those rules that might be advantageously applied.

We see the QA4 theorem prover working at the same level as a human mathematician, and a finished QA4 proof should read like a proof in a mathematical textbook. To illustrate this point we present a fairly difficult theorem, and a protocol of the projected QA4 proof procedure applied to this theorem. The following discussion presents only the "correct" branch of the hypothetical QA4 solution. A problem solving strategy that would generate this solution, among others, is described in the next section of this note.

The theorem to be proved arises in a program-synthesis problem. We are given a recursive program to compute the Fibonacci sequence 1,1,2,3,5,8, ... in which each term is the sum of the preceding two terms. The program we are given is

$$\text{fib}(x) = \text{if } x \leq 1 \text{ then } 1 \text{ else fib}(x - 1) + \text{fib}(x - 2) \quad .$$

This program is grossly inefficient, requiring many redundant recursive computations of the function on the same argument. We would like to construct an equivalent iterative program.

Of the many possible QA4 rules of inference, the following are useful in this problem.

- (1) Induction (Going-Up Iterative⁴): To prove a theorem of the form $(\forall x)P(x)$, where x is a natural number, prove $P(0)$ and prove $(\forall x)P(x) \supset P(x + 1)$.
- (2) Resolution: The equivalent of Robinson's rule,⁵ but expressed in terms of QA4 expressions with quantifiers.
- (3) Partial Evaluation: Take a function that is defined in the system, and expand it according to its definition. For example, replace $\text{fib}(x + 2)$ by $\text{fib}(x + 1) + \text{fib}(x)$). The rule especially applies to expressions of the form $f(a)$ or $f(x + a)$, where a is a constant.
- (4) Conditional Split: Replace an expression of the form $\text{if } P \text{ then } Q \text{ else } R$ by $(P \supset Q) \wedge (\neg P \supset R)$.
- (5) Conditional Derivation: To prove a theorem of form $P \supset Q$, assume P and prove Q .

- (6) \wedge -Split: To prove a theorem of form $P \wedge Q$, prove P and prove Q . When an assertion of form $P \wedge Q$ is made, assert P and assert Q .
- (7) Functional Split: To prove a theorem of form $(\exists z) z = f(t_1, \dots, t_n)$, prove a theorem $(\exists z) z_1 = t_1 \wedge \dots \wedge (\exists z_n) z_n = t_n$.
- (8) Equality: To prove a theorem of form $t_1 = t_2$, where the t_i are terms, replace the existentially quantified variables of the t_i so that the two resulting terms are identical.
- (9) Change of Variables: Replace an expression of form $(\forall x) [x \geq a \supset P(x)]$, where x is a natural number, by $(\forall x) [P(x + a)]$ (replacing x by $x - a$).
- (10) Simplification: Replace $1 + 1$ by 2 , $0 \cdot X$ by 0 , and make other such improvements.

These rules are roughly stated; for example, the forms that \wedge -split, conditional split, and the equality rule are applied to may have certain quantifiers. In practice these rules would be separate, complex programs in the QA4 language.

Now let us examine the behavior of the system when faced with the program synthesis problem. We first assert

- (11) Assert $\text{fib} = \lambda x \text{ if } x \leq 1 \text{ then } 1 \text{ else } \text{fib}(x - 1) + \text{fib}(x - 2)$.

(12) Assert $(\forall x) (x \leq 1 \supset \text{fib}(x) = 1)$

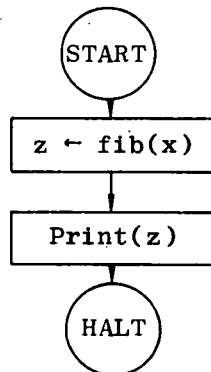
and

(13) Assert $(\forall x) x \geq 2 \supset \text{fib}(x) = \text{fib}(x - 1) + \text{fib}(x - 2)$.

To produce (13) the system used the simplifier to replace $\neg(x \leq 1)$ by $x \geq 2$; we will not always mention the actions of the simplifier explicitly. We then give the system the goal

(14) Construct an iterative program that satisfies the input-output relation, $z = \text{fib}(x)$, where x is the input and z is the output, and fib is not taken to be "primitive."

The condition that fib not be primitive means that fib is not permitted to appear in the iterative program. This restriction is intended to prevent the system from producing the following iterative program.



(This program is correct, iterative, and every bit as in efficient as the original recursive program.)

When the system is given this program-synthesis goal, it may transform it into a theorem-proving goal by using a standard technique.¹

Thus, it produces the new goal

(15) Prove $(\forall x)(\exists z) z = \text{fib}(x)$.

From its collection of inference rules, the system selects those that seem relevant to the proof of this theorem. These are induction, equality, and resolution (against 12 or 13). Induction is an expensive routine; we will defer trying it until we have explored the other possibilities. Equality tries to substitute $\text{fib}(x)$ for z ; however, the stipulation that fib is not primitive prevents that substitution from being made; otherwise, the proof would be concluded and the trivial program above would be produced. In this case, however, the equality rule fails. "Resolution" of (15), with (12) produces

(16) Prove $(\forall x) x \geq 2 \supset (\exists z) z = \text{fib}(x)$.

This goal is more attractive than the original goal (15) because it is a special case of (15); (16) is the consequent of (15). Therefore, the attention of the system is focussed on (16), and work on (15), including application of the induction rule, is delayed. The system then selected those rules that seem relevant to the proof of (16). The rules selected include change of variables (9), conditional derivation (5), and induction. Change of variables is applied before the other rules, producing a new goal

$$(17) (\forall x)(\exists z) z = \text{fib}(x + 2).$$

The form of (17) suggests the immediate application of the partial evaluation rule (3). This produces (with simplification)

$$(18) \text{ Prove } (\forall x)(\exists z) z = \text{fib}(x - 1) + \text{fib}(x).$$

This goal is in the proper form for functional splitting (7).

The new goal,

$$(19) \text{ Prove } (\forall x)[(\exists z_1) z_1 = \text{fib}(x + 1) \wedge (\exists z_2) z_2 = \text{fib}(x)],$$

is produced. Although the form of this expression suggests \wedge -splitting, this tack quickly proves to be a dead end: of the two goals produced,

$$(20) \text{ Prove } (\forall x)(\exists z_1) z_1 = \text{fib}(x + 1) \text{ and}$$

$$(21) \text{ Prove } (\forall x)(\exists z_2) z_2 = \text{fib}(x),$$

the second proves to be identical to the original goal (15). Since both these goals must be achieved in order that (19) be achieved, both (20) and (21) are discarded. Having exhausted the other possibilities, the system ventures to try induction on (19). The two new goals generated are:

$$(22) \text{ Prove } (\exists z_1) z_1 = \text{fib}(1) \wedge (\exists z_2) z_2 = \text{fib}(0), \text{ and}$$

$$(23) \text{ Prove } (\forall x) [((\exists z_1) z_1 = \text{fib}(x + 1) \wedge (\exists z_2) z_2 = \text{fib}(x)) \supset ((\exists z'_1) z'_1 = \text{fib}(x + 2) \wedge (\exists z'_2) z'_2 = \text{fib}(x + 1))] .$$

Both these goals must be achieved if the theorem is to be proved. The system considers the first goal first. The most appropriate rule to be applied is \wedge -split, which produces two new goals,

(24) Prove $(\exists z_1) z_1 = \text{fib}(1)$ and

(25) Prove $(\exists z_2) z_2 = \text{fib}(0)$,

both of which must be achieved. Partial evaluation applies to both goals, producing

(26) Prove $(\exists z_1) z_1 = 1$ and

(27) Prove $(\exists z_2) z_2 = 1$.

Then the equality rule is applied to each of these goals with success, so that (22) has been achieved. Attention now focusses on (23). Conditional derivation (5) allows us to make the assumption

(28) Assert $(\exists z_1) z_1 = \text{fib}(x + 1) \wedge (\exists z_2) z_2 = \text{fib}(x)$, and

create the goal

(29) Prove $(\exists z'_1) z'_1 = \text{fib}(x + 2) \wedge (\exists z'_2) z'_2 = \text{fib}(x + 1)$.

The \wedge -split rule, applied to the assertion (28), produces two new statements,

(30) Assert $(\exists z_1) z_1 = \text{fib}(x + 1)$ and

(31) Assert $(\exists z_2) z_2 = \text{fib}(x)$.

The same rule, applied to the goal (29), results in the establishment of two other goals

(32) Prove $(\exists z'_1) z'_1 = \text{fib}(x + 2)$

and

(33) Prove $(\exists z'_2) z'_2 = \text{fib}(x + 1)$,

both of which are to be achieved.

The resolution rule applies between goal (33) and assertion (30) resulting in a success. Partial evaluation, applied to goal (32) constructs:

(34) Prove $(\exists z'_1) z'_1 = \text{fib}(x + 1) + \text{fib}(x)$.

As before, function splitting produces

(35) Prove $(\exists z_3) z_3 = \text{fib}(x + 1) \wedge (\exists z_4) z_4 = \text{fib}(x)$,

and \wedge -split produces

(36) Prove $(\exists z_3) z_3 = \text{fib}(x + 1)$ and

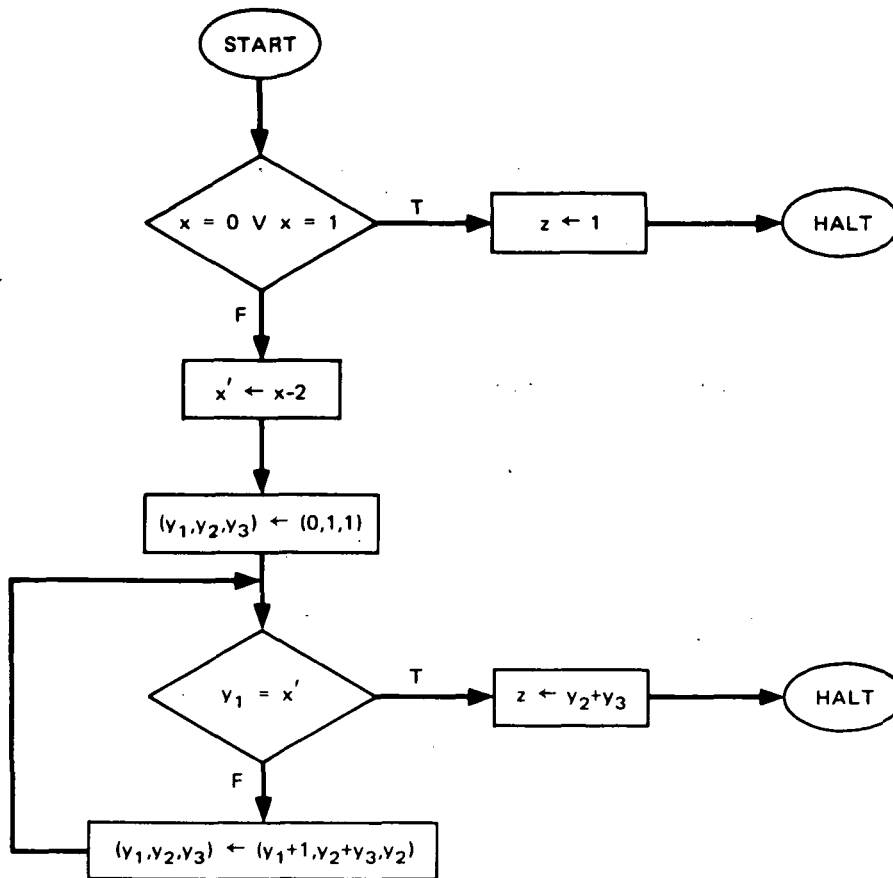
(37) Prove $(\exists z_4) z_4 = \text{fib}(x)$.

These goals resolve with assertions (30) and (31) respectively, completing the proof.

We have included mostly those steps in the search that actually did lead to the proof. The system would examine some of the false paths too, although it does not rely on blind search and discontinues a line of reasoning when another appears more profitable.

Program synthesis techniques allow us to produce the program illustrated in Figure 1, from the proof.⁴ This program turns out to be far more efficient than the original recursive program.

In this section we have discussed the behavior of a problem solver without specifying a mechanism that exhibits this behavior. In the next section we outline a system capable of carrying out such reasoning.



TA-8721-2

FIGURE 1 ITERATIVE FIBONACCI PROGRAM

IV The QA4 PROBLEM SOLVER

This section gives an overview of the goals, overall structure, and flow of control of the QA4 problem solver.

A. Goals

- The problem solver should be easy to guide with intuitive knowledge about various forms of problem solving. If we run a proof, for example, and we see the problem solver doing an obviously stupid thing, then it should be possible to modify the proof strategy or give additional information in an easy way so that the system does not make

the same errors in a second run of the problem. Thus, the problem solver should also be easily modifiable.

- A large body of pragmatic information in the system
- A natural and compact formulation not only of goal statements but also of strategies in a unified language.

For example, we would not write the theorems to be proved in first-order predicate calculus while writing strategies in LISP.

B. Statements

The system is given information with four sorts of statements:

- Goal statements: e.g., Prove $(\forall x)(\exists z) z = \text{fib}(x)$
- Assertions: e.g., factorial = λx if $x = 0$ then 1 else $x \cdot \text{factorial}(x - 1)$
- Eval rules: e.g., change of variables $(\forall x). x \geq a \supset P(x)$ transforms to $(\forall x)P(x + a)$
- Strategies: e.g., a linear equation solver.

The goal statements and assertions are analogous to the theorems and axioms of a resolution-type theorem prover. The eval rules and strategies are expression transformation rules.

An eval rule is a single-expression transformation rule. It takes an input expression, matching a pattern given in the first half

of the eval rule, and transforms it under given conditions (when a predicate is true) into an output expression according to the second half of the eval rule.

A strategy is a program made up of control statements, eval rules, and other strategies. The program tells how to apply several transformations, sequentially or in parallel, for example.

C. Basic Method

The system is goal-directed. A problem entered in the system is the first goal statement. The system tries to find eval rules and strategies that may aid in achieving the goal. From these rules it constructs a single strategy associated with the goal. This strategy is applied to the goal; if this strategy does not succeed at once, the system may create one or more subgoals. In the same way, subgoals are given associated strategies, which control their processing.

The eval rules and strategies relevant to a given goal or assertion are selected by the "filter."

D. The Filter

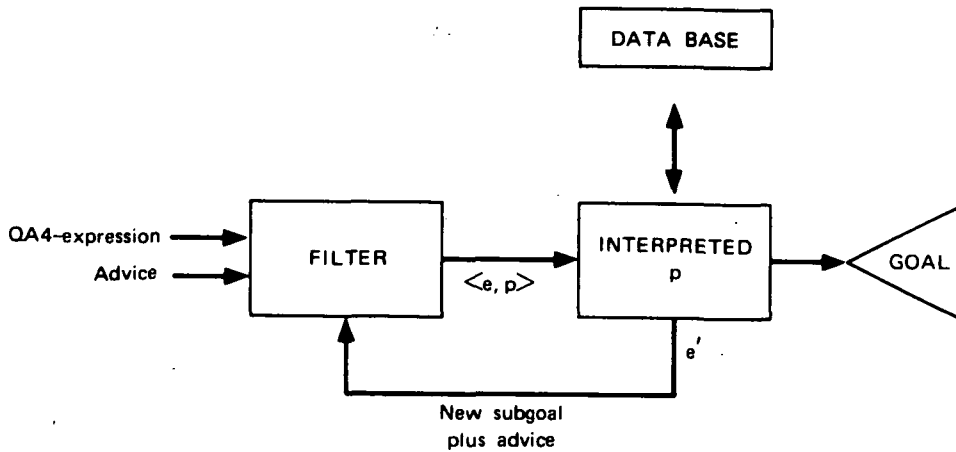
The filter is a program that analyzes expressions and the associated semantic and pragmatic information kept on the expression's property list. The filter's main task is to find in an efficient way all eval rules, strategies, and typed-in pragmatics applicable to

(matching with) a given expression. After it has found the relevant information, a combined strategy is put together, put on the property list of the expression, and given to the interpreter.

E. How Statements are Processed

Let us see how the system processes each sort of QA4 input statement. First, consider the case of an assertion given to the filter. An assertion must be entered in the data base of the problem solver. It is possible that whenever an assertion of a certain form (matching a given pattern) is made, other assertions also should be made. We can give a great number of this sort of rules in the form of eval rules. An example is the conditional-split rule, which is applicable to the assertion $\text{fib} = \lambda x_1 . \text{if } \dots$ in the example of Section IV. Two additional assertions must be made according to this rule. Matching rules are found by the filter. A strategy is made up and interpreted that puts the initial assertion and the assertions discovered by the filter in the data base.

In the case of a goal statement, an expression is given to the filter together with advice. For example, the goal statement "Prove $z = \text{fib}(x)$ " is given to the filter, together with constraints and advice, such as: "the given expression is an input/output relation, this is a program-writing problem, write an iterative program." The filter tries to find the relevant eval rules and strategies with the information



TA-8721-1

residing in the filter. It will do some pattern matching to find relevant expression transformation rules, and use the constraints and advice given, together with the goal statement, in the search for the right rules. In the example, the filter puts the strategy "try the theorem-proving approach" together. This strategy creates the new subgoal "Prove $(\forall x)(\exists z) z = \text{fib}(x)$." The strategy gives the subgoal, together with the advice "try only techniques that give iterative solutions," to the filter. Now the whole procedure will be repeated until success is achieved and the goal can be proved true.

The filter is changed by entering new eval rules and strategies. The front end of an eval rule (a pattern) will get its proper place

among the already collected patterns in the filter; e.g., the eval rule change of variables will cause the filter to be updated with the pattern $(\forall x)x \geq a \supset P(x)$. When an expression of that form is passed through the filter, the change-of-variables rule will be selected.

F. How Problems are Solved

All strategies, eval rules, the filter, a simple monitor, and other high-level programs of the problem solver are written in the QA4 language. For this language, a simple LISP-like EVAL is being written.

The flow of control in the system is governed by strategies, interpreted by a simple monitor. Strategies are put on property lists of expressions according to certain conventions. The task of the monitor is to interpret strategies under a set of conventions. The monitor also hands expressions to the filter and utility functions; for example, a function that puts typed-in information about a problem statement on the property list of this expression. The monitor interprets the control functions and in general connects the complex of strategies and system functions. The task of the monitor is, however, a mechanical task: All "cleverness" resides in the strategies.

The situation of a strategy creating one subgoal can get more complex when more eval rules or strategies are applicable; e.g., in the example of the fib function: Try partial evaluation, resolution, or induction. Now the system can work on one subgoal, but should not give

up on the other subgoals. It could work for a time on the goal generated by the partial evaluation but then decide that the goals are getting worse (compared with the original) and try the induction step.

To be able to work in such a fashion, a set of functions for controlling strategies are available. They will be all realized with a simple coroutine mechanism that makes use of the contexts as described in Section III.

G. Control Functions

To give the flavor of the control functions, some are described below. A strategy can create two or more goals and ask the problem solver to prove them all. An example is induction, in which two subgoals (the zero case and the step case) must be proved true. The system uses for this purpose the AND statement (AND set strategy). All the strategies in the set are run in parallel, and the relative speed of each program is controlled by the strategy. Sometimes it is necessary for a program in the set to communicate with the controlling strategy. For example, the program sees its progress is poor and wants to give this information to the controlling strategy of the AND, so that another program in the set can be given a turn or other action can be taken. For this purpose a program (strategy) can use the WAIT statement (WAIT x). The value of x is given to the strategy associated with the AND and the calling program is suspended. The OR statement (OR set strategy) operates in a similar

way. For example, in the Fibonacci problem, three alternative rules are proposed for the goal $(\forall x)(\exists z) z = \text{fib}(x)$: induction, resolution, and the equality rule. These rules are combined by an OR statement and equality is tried first, but fails. Now resolution is selected by the strategy associated with the OR statement. Induction is only tried when the resolution strategy fails or produces poor results, in which case a return to the OR statement is made. In the case of the Fibonacci example the resolution was successful.

H. Advice to the System During a Proof

The problem solver is able to take advice during a proof.

A natural point to do this is whenever a strategy calls the filter and gives a new goal (or new goals) to be analyzed. We can envision among others two ways of giving advice:

- (1) Changing a strategy, mainly strategies controlling AND and OR statements; and
- (2) Supplying a new strategy in the set of an AND or OR, which gives rise to a new subgoal.

REFERENCES

1. D. Hilbert and W. Ackermann, Principles of Mathematical Logic, pp. 152-163 (Chelsea Publishing Company, New York, New York 1950).
2. L. J. Chaitin et al., "Research and Applications--Artificial Intelligence," Interim Scientific Report, Section V, "Long Term Problem Solving," Contract NAS12-2221, SRI Project 8259, Stanford Research Institute, Menlo Park, California (April 1970).
3. R. E. Fikes, "A LISP Implementation of BIP," AI Group Technical Note No. 22, Stanford Research Institute, Menlo Park, California (February 1970).
4. Z. Manna and R. Waldinger, "Towards Automatic Program Synthesis," AI Group Technical Note 34 (July 1970). [Submitted for publication in Collection of Lecture Notes in the Symposium on the Semantics of Algorithmic Languages, Erwin Engeler, ed., Springer Verlag.] [Also submitted for publication in the Communications of the Association for Computing Machinery.]
5. J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," Journal of the Association for Computing Machinery Vol. 12, No. 1 (January 1965).

APPENDIX F

SOME CURRENT TECHNIQUES FOR SCENE ANALYSIS

October 1970

SOME CURRENT TECHNIQUES FOR SCENE ANALYSIS

by

Richard O. Duda

Artificial Intelligence Group

Technical Note 46

SRI Project 8259

This research is sponsored by the Advanced Research
Projects Agency and the National Aeronautical and
Space Administration under Contract NAS 12-2221.

I INTRODUCTION

The purpose of the visual system is to provide the automaton with important information about its environment, information about the location and identity of walls, doorways, and various objects of interest. By adding new information to the model, the visual system gives the automaton a more complete and accurate representation of its world. The role of vision is not independent of the state of the model. If the automaton has entered a previously unexplored area, the visual scene must be analyzed to add information about the new part of the environment to the model. In this situation, the model can provide so little assistance that it is often not referenced at all. On the other hand, if the automaton is in a thoroughly known area, the role of vision changes to one of providing visual feedback to correct small errors and verify that nothing unexpected has happened. In this situation, the model plays a much more important role in assisting and actually guiding the analysis.

Until recently our attention has been directed primarily at the general scene-analysis problem. Every picture was viewed as a totally new scene exposing completely unknown area. More recently we have addressed the problem of using a complete, prespecified map of the floor area to update the automaton's position and help in tasks such as going through a doorway. Another use of this kind of visual feedback would be the monitoring of objects being pushed.

In trying to solve these problems, we have tended to take one or the other of two extreme approaches. Either we tried to develop general methods that can cope with any possible situation in the automaton's

world, or we tried to exploit rather special facts that allow an efficient special-purpose solution. The first approach involves the more interesting problems in artificial intelligence, but it provides more capabilities than are needed in many situations, and provides them at the cost of relatively long computation times. The second approach provides fast and effective solutions when certain (usually implicit) preconditions are satisfied, though it can fail badly if these conditions are not met. Eventually, of course, some combination of these two approaches will be needed, since the automaton actually operates in a partially known world, rather than one that is completely unknown or completely known. However, we have decided to concentrate on these two extreme situations before addressing the intermediate case. The remainder of this note describes the current status of our work in these areas.*

II REGION ANALYSIS

A. The Merging Procedure

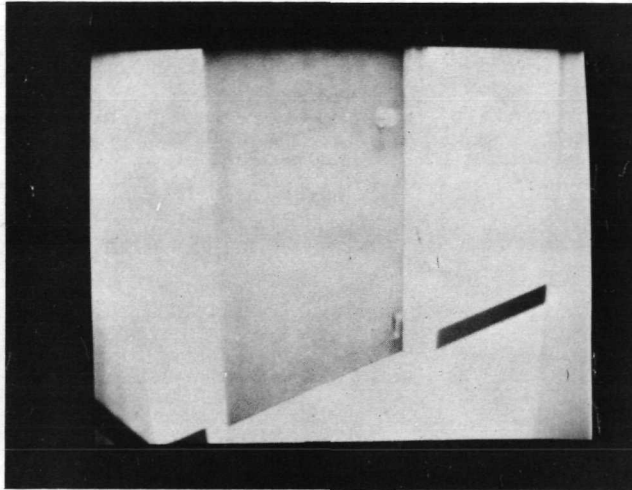
Our work in general scene analysis is based on dividing the picture into regions representing walls, floors, faces of objects, etc. The basic approach has been described in detail elsewhere,³ and only a brief summary will be given here. The procedure begins by partitioning the digitized image into elementary regions of constant brightness. This usually produces many small, irregularly shaped regions that are fragments of more meaningful regions. Two heuristics are used to merge

*Our earlier work in scene analysis is described in Reference 1. Additional information on more recent work is contained in References 2-5. References are listed at the end of this report.

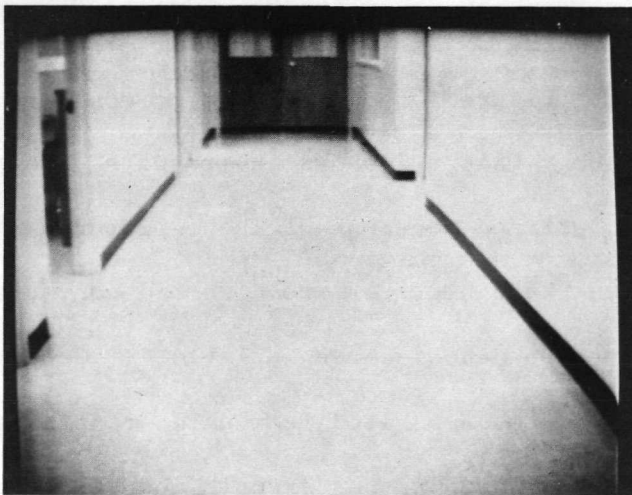
these smaller regions together. Both of these heuristics operate on the basis of fairly local information, the difference in brightness along the common boundary between two neighboring regions. The heuristics are not infallible; they can merge regions that should have been kept distinct, and they can fail to merge regions that should have been merged. However, they reduce the picture to a small number of large regions corresponding to major parts of the picture, together with a larger number of very small regions that can usually be ignored.

The effect of applying these heuristics is best described through the use of examples. Figure 1 shows television monitor views of three typical corridor scenes. Figure 2 shows the results of applying the merging heuristics to digitized versions of these pictures. The boundaries of the regions in these pictures are directed contours, and can be traced using the correspondences shown in Table I. Generally speaking, important regions can be separated from unimportant regions purely on the basis of size. Figure 2a, for example, contains four large, important regions. Three of them are directly meaningful (the door, the wall to the right, and the baseboard), and the fourth is the union of two important regions (the floor and the wall to the left). An inspection of Figure 2b shows similar results. Figure 2c shows the result of applying the technique to a complicated scene; while some useful information can be obtained, the resolution available severely limits the usefulness of the results.

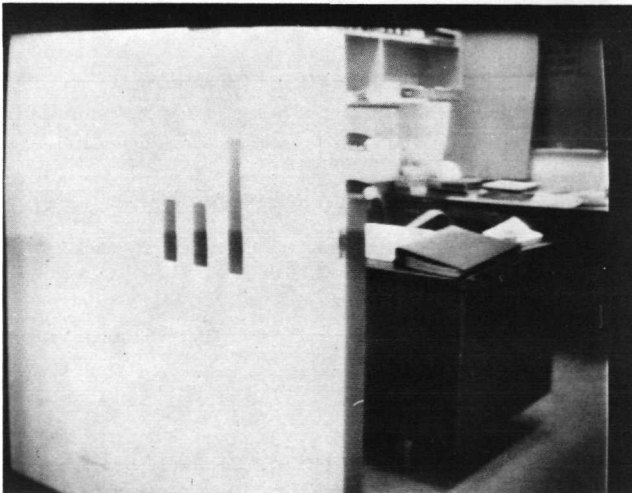
Our only complete scene-analysis program is oriented toward identifying boxes and wedges, objects with triangular or rectangular



(a) DOOR



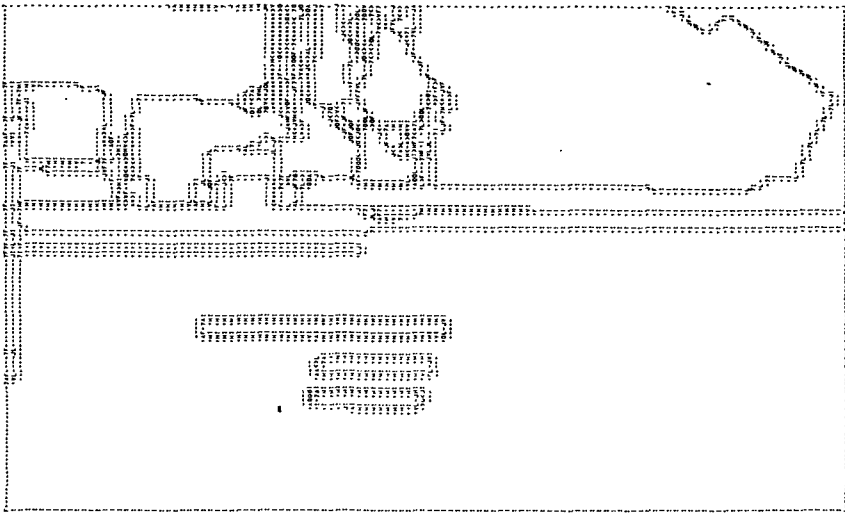
(b) HALL



(c) OFFICE WITH SIGN

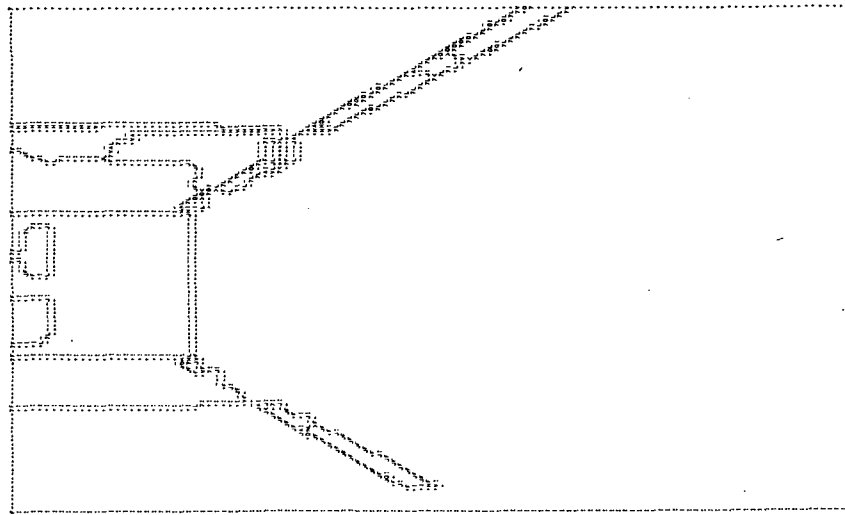
TA-8259-20

FIGURE 1 THREE CORRIDOR SCENES

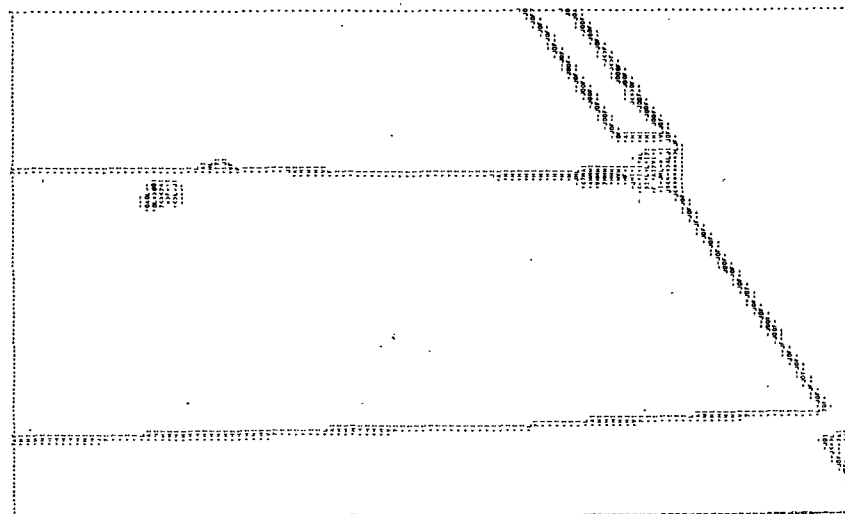


(c) OFFICE WITH SIGN

TA-8259-21



















(b) HALL



(a) DOOR

FIGURE 2 RESULTS OF MERGING HEURISTICS

TABLE I CORRESPONDENCE BETWEEN
BOUNDARY SEGMENT CONFIGURATIONS
AND CHARACTERS USED IN PRINTOUT

CONFIGURATION	CHARACTER	CONFIGURATION	CHARACTER
			!
	-		└
	↑		H
	J		V
	↑		F
	=		>
	7		A
	>		O

TA-8259-24

faces, in a simple room environment.³ For this task, we begin by fitting the boundaries of the major regions by straight lines. Regions are identified as being part of the floor, walls, baseboards, and faces of objects by such properties as shape, brightness, and position in the picture. Objects are identified by grouping neighboring faces satisfying some of the simpler criteria used by Guzman.⁶ In the process, certain errors caused by incorrect merging are detected and corrected. We have yet to complete a similar analysis program for the conditions encountered in corridor scenes. However, we have investigated the problem of obtaining a scene description that is internally consistent; the next section describes the analysis approach for this problem.

B. A Procedure for Scene Analysis

If we assume temporarily that the merging heuristics have succeeded in the sense that all of the large regions are meaningful areas, then the only basic problem remaining is the proper identification of each region. Examination of the corridor pictures indicates the need to be able to identify a number of different region types, including the following:

- (1) Floor
- (2) Wall
- (3) Door
- (4) Door jamb
- (5) Object face
- (6) Baseboard
- (7) Baseboard reflection
- (8) Sign*
- (9) Window

*By "sign" we mean a dark vertical bar on the wall used, as illustrated in Figure 1c, to identify an office.

- (10) Clock
- (11) Doorknob
- (12) Thermostat
- (13) Power outlet
- (14) Automaton.

Each of these regions has certain properties which tend to characterize it uniquely. For example, the floor region is usually large, bright, and near the bottom of the picture. However, most regions can be identified with greater confidence if the nature of their neighbors is considered as well. Thus, the presence of a baseboard or baseboard reflection at the top of a region almost guarantees that the region is the floor; conversely, the presence of wall area immediately above a region guarantees that it can not be a baseboard reflection. If regions are identified without regard to how that choice affects the overall scene description, the chance for error is increased. Moreover, the resulting description can be nonsensical.

Many, though by no means all, of the relations between types of regions relate to neighboring regions. Table II indicates those types of regions that can and cannot be legal neighbors. We can easily add to this further restrictions, such as the fact that the baseboard must have the wall as a neighbor along its top edge. These are some of the important known facts about the general nature of the automaton's environment. The problem is to use facts such as these to aid in the analysis of the scene.

One approach to solving this problem is to use these facts as constraints to eliminate impossible choices. Suppose that each significantly large region in the picture is tentatively classified

TABLE 11 REGIONS THAT ARE LEGAL NEIGHBORS

	FLOOR	WALL	DOOR	DOOR JAMB	OBJECT FACE	BASEBOARD	BASEBOARD REFLECTION	SIGN	WINDOW	CLOCK	DOORKNOB	THERMOSTAT	POWER OUTLET	AUTOMATON
FLOOR		+	+	+	+	+	+							
WALL	+	+	+	+	+	+		+	+	+	+	+	+	+
DOOR	+	+		+	+	+			+		+			+
DOOR JAMB	+	+	+		+	+					+			+
OBJECT FACE	+	+	+	+	+	+	+	+	+		+	+	+	+
BASEBOARD	+	+	+	+	+	+	+						+	+
BASEBOARD REFLECTION	+				+	+	+							+
SIGN		+			+									+
WINDOW		+	+		+									
CLOCK		+												
DOORKNOB		+	+	+	+									
THERMOSTAT		+			+									
POWER OUTLET		+			+	+								+
AUTOMATON	+	+	+	+	+	+	+	+					+	

TA-8259-25

on the basis of the attributes of that region alone. Suppose further that a score is computed for each region that measures the degree to which it resembles each region type.* For any selection of names for regions, we can define the score for the resulting description as the sum of the individual scores. Then, we can analyze the scene by trying to find highest scoring legal selection of region names. With no loss in generality and some gain in convenience, we can work with the losses incurred by selecting other than the highest scoring choice. In terms of losses, we want the legal description having the smallest overall loss.

This problem is basically a tree-searching problem. The start node of the tree corresponds to the first region selected for naming. The branches emanating from that node correspond to the possible choices of names for that region. A path through the tree corresponds to a unique labeling of the picture. Thus, if there are N possible region names and R regions, there are potentially N^R possible paths through the tree. Each path passes through $R+1$ nodes from the start node to the terminal node. Every terminal node has a loss value, which is the sum of the losses incurred for the choices along the path to that node. A goal node is a terminal node corresponding to a complete, legal scene description. We seek the goal node with the smallest overall loss.

This is a standard problem in tree searching, and optimum search procedures are known. Assume that some choices have been made for some of the regions so that we have a partially expanded tree.

* This score might be interpreted as the logarithm of the probability that the given region is of the indicated type.

Using the Hart-Nilsson-Raphael terminology,⁷ some of the terminal nodes of this tree are open nodes, candidates for further expansion. Each open node has an associated loss \hat{g} , the sum of the losses from the start node to that node. If we assume that there is no reason to believe that zero-loss choices cannot be made from that node on, then the optimal search strategy is to expand that open node having the minimum \hat{g} .

To expand a node, we must select a region not previously considered and examine the possible choice for that region, ruling out any choices that are not legal. Different strategies can be used for selecting the next region. It seems advantageous to ask it to be a neighbor of the regions selected previously, since this maximizes the chance of detecting illegalities. In general, we will have several neighbors for candidate successors. Of these, it seems reasonable to select the one having the highest score, under the assumption that the first choice name for this region is most likely to be correct.

After a region has been selected, it is necessary to examine the choices one can make for its name to see which ones are legal. If we limit ourselves to pairwise relations between neighboring regions, we need merely compare each choice with previously made choices on the path to this point and test each for legality.* The node expanded is removed from the list of open nodes, the resulting new nodes are added, and the process is repeated until the algorithm selects a goal node for further expansion. This is our final result, a legal scene description having the minimum loss.

* When an illegality is found, that choice is deleted. One can argue that few relations are so strong as to be absolutely illegal, and an alternative approach would be to introduce various additional losses for the different observed relations.

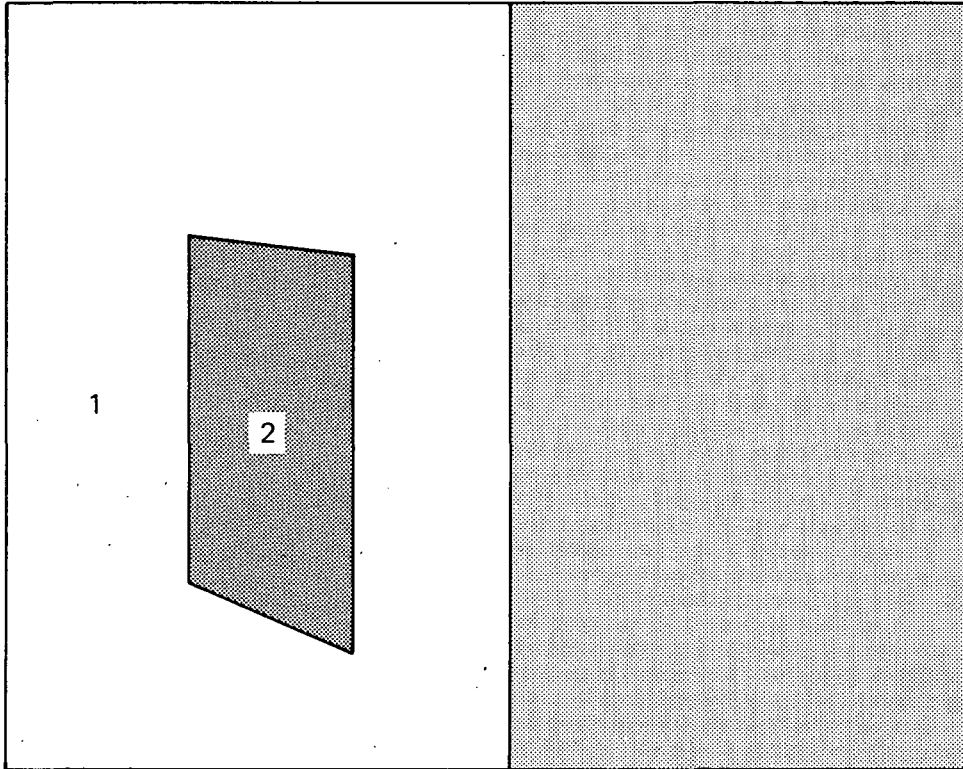
C. Examples

The following examples serve to illustrate the action of this scene-analysis procedure. Consider first the simple scene shown in Figure 3. For simplicity, we assume that there are only five types of allowed regions--floor, wall, door, baseboard, and sign. Consider Region 1. On the basis of its brightness, size, vertical right boundary, and possession of a hole, it should receive a high score as a wall, and lower scores as floor, door, sign, and baseboard. Region 2 might, perhaps, score highest as a door, and so on. Thus, the following table of scores, although purely imaginary, is not unreasonable. Missing entries correspond to scores too low to be seriously considered.

Type Region	Floor	Wall	Door	Base- board	Sign
1	5	6	2		
2			7	1	5
3	3	3	5		1

The following table gives equivalent information in terms of the losses associated with each choice.

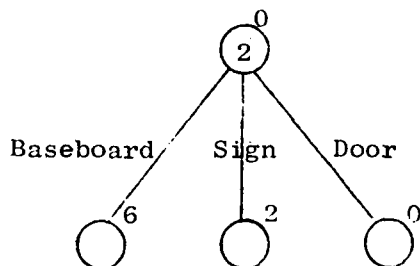
Type Region	Floor	Wall	Door	Base- board	Sign	Max Score
1	1	0	4			6
2			0	6	2	7
3	2	2	0		4	5



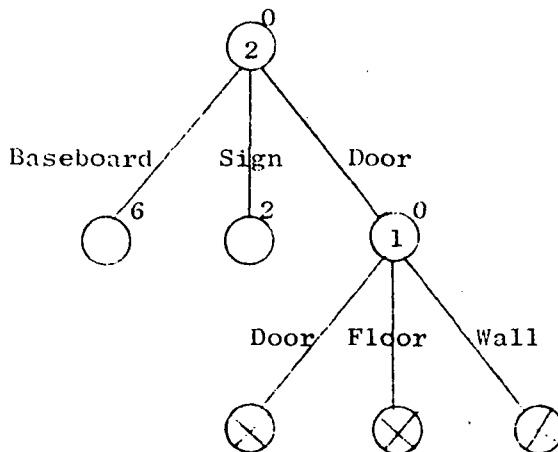
TA-8259-26

FIGURE 3 A SIMPLE SCENE

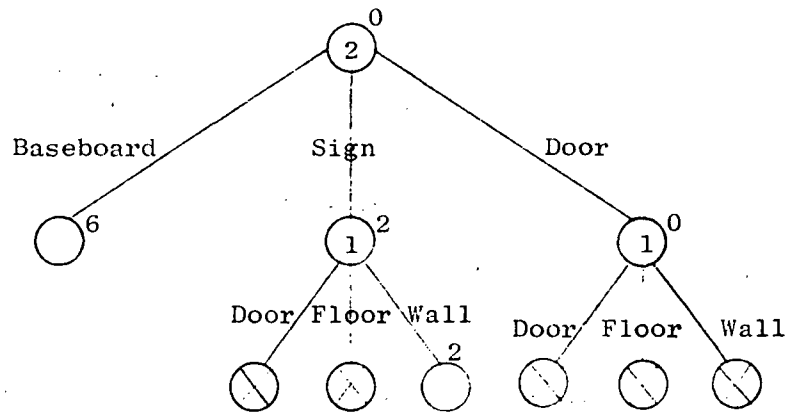
Let us use our tree-searching algorithm to obtain the minimum-loss, legal description of this scene. Initially the successor function is unconstrained by neighbor restrictions, and selects Region 2 merely because it has the highest score. At this point, all of the choices for Region 2 are legal, and the tree has three open nodes; the numbers shown next to each node give the loss accumulated in reaching that part of the tree.



The search algorithm requires that the open node having the least loss be expanded next, which corresponds to tentatively calling Region 2 a door. The successor function finds only one neighbor to choose from, Region 1, and considers its alternatives: wall, floor, and door. None of these choices is a legal neighbor surrounding Region 1, and hence all are rejected. Thus, this open node has no successors.

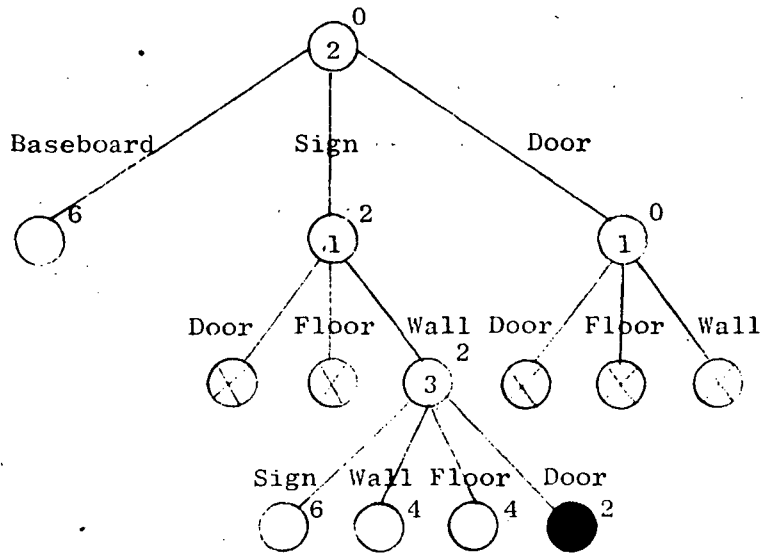


Returning to the choices for open nodes, Region 2 is tentatively called a sign. The successor function again selects Region 1, and this time finds one legal successor, the wall.* The loss associated with this choice is 0, and the overall loss is 2. The list of open nodes still contains two members.



The search algorithm selects the open node with loss 2, and the successor function has only Region 3 to select from. All of the choices for Region 3 are all legal with respect to calling Region 2 a sign and Region 1 a wall. The least loss results from calling Region 3 a door, and the scene analysis is completed.

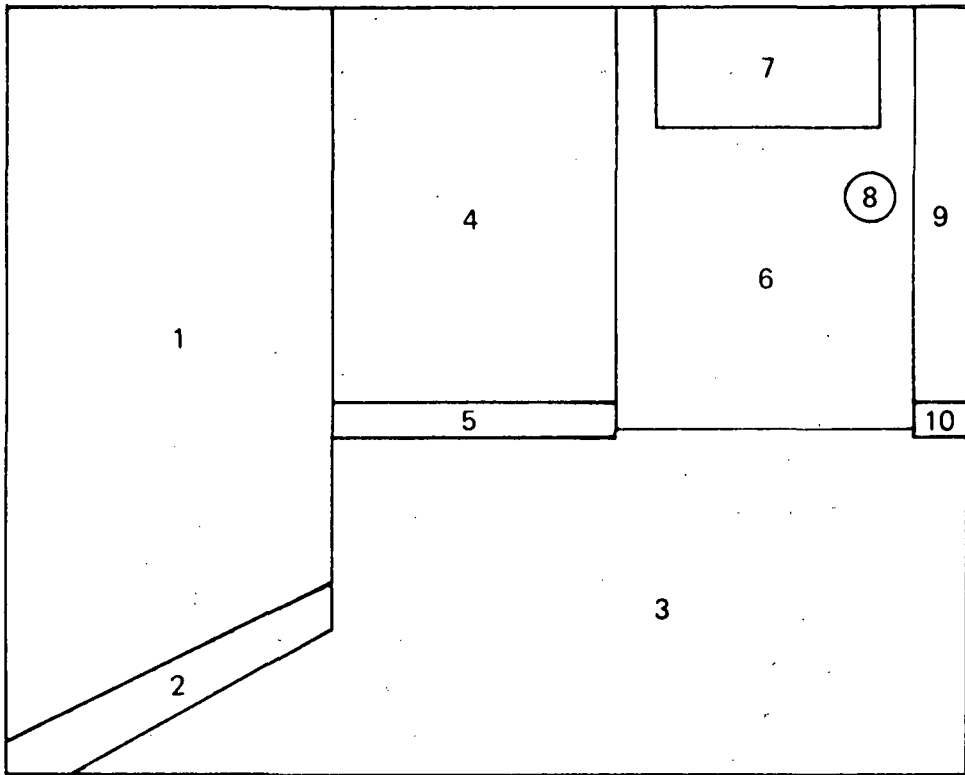
* Note that our successor function will always produce a tree with R+1 levels. At any level, the same region will always be selected by the successor function. The actual successors, however, will be limited by the legality requirement.



A somewhat more realistic example involving 10 regions and 14 region types is illustrated in Figure 4. Table III gives the hypothetical scores. Based on these scores alone, half of the regions would be incorrectly identified. Figure 5 shows the tree produced by the search algorithm. The development of this tree is too complicated to describe in detail. It should be noted, however, that considerable backtracking occurred because a low-scoring third choice was needed for Region 8, the doorknob. Whether or not this can be circumvented without causing other problems is not known.

D. Remarks

To date, this procedure has only been used on some hypothetical examples. We have modified a general tree-searching program to adapt it to some special characteristics of this problem. However, we have not started the important task of writing programs to measure characteristics of regions and to use these characteristics to produce recognition scores.



TA-8259-27

FIGURE 4 A MORE COMPLICATED SCENE

TABLE III HYPOTHETICAL REGION SCORES

TYPE	REGION									
	1	2	3	4	5	6	7	8	9	10
FLOOR	1		11			2				
WALL	7		3	5		5			4	
DOOR	3			6		6			3	
DOOR JAMB									6	
OBJECT FACE							6			
BASEBOARD		5			9					3
BASEBOARD REFLECTION		7			5					
SIGN		1								6
WINDOW	1			2			8			
CLOCK								1		
DOORKNOB								2		
THERMOSTAT								6		
POWER OUTLET								3		4
AUTOMATON										

TA-8259-29

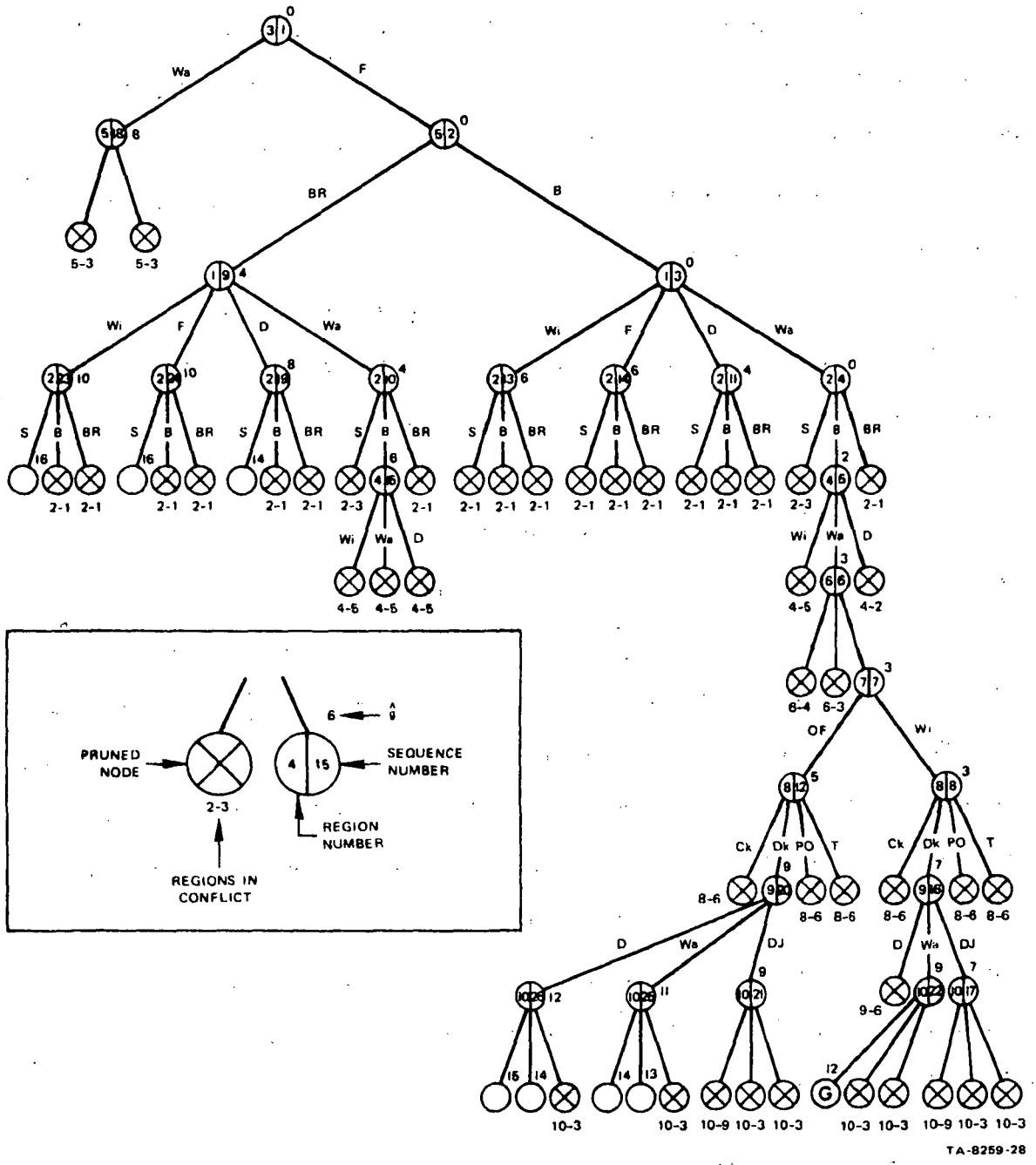


FIGURE 5 THE ANALYSIS TREE

In addition, we have not implemented any legality conditions beyond the simple conditions given in Table II.

This approach to scene analysis has several potential advantages. It is not necessary to identify every region correctly at the outset to obtain a correct analysis, provided that the "syntactic" rules are sufficiently complete. By providing a limit on the allowable loss, a partial scene description can be obtained that may be useful even though incomplete. Perhaps most important, the operations of merging, feature extraction, classification, and analysis are clearly separated, allowing fairly independent modification and improvement. In particular, the general knowledge about the environment can be expressed explicitly as rules for legal scenes, and if the environment is changed it is possible to confine the program changes to modifying these rules.

One of the major problems with this approach is the lack of an obvious way to detect erroneous regions, regions that are fragments of or combinations of meaningful regions. We are currently working on this problem, since progress toward its solution is needed before implementation of this system can be begun. Another problem is that it is not clear how specific information contained in the model can be used to guide the analysis. This problem of working in a world that is neither completely known nor completely unknown is one of the major unsolved problems in visual scene analysis.

III LANDMARK IDENTIFICATION

When the environment is completely known, the visual system can provide feedback to update the automaton's position and orientation.

The x-y location of the automaton and its orientation θ can be determined uniquely from a picture of a known point and line lying in the floor.* Such distinguished points and lines serve as landmarks for the automaton. This section describes our present program that uses concave corners, convex corners, and doorways as landmarks to update position and orientation.

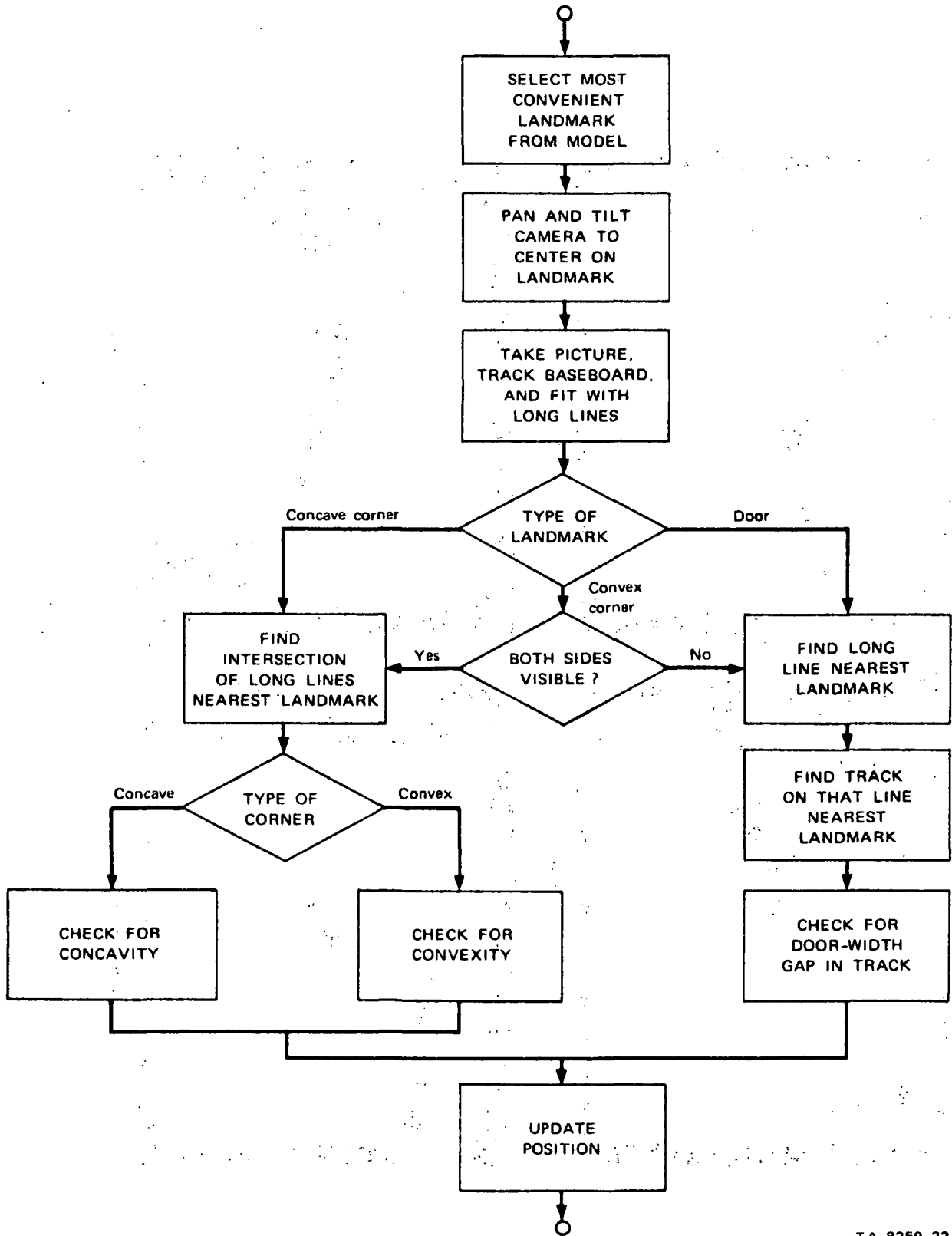
A flowchart outlining the basic operations of this program is shown in Figure 6. The program begins by selecting a landmark from the model that should be visible from the automaton's present position; if more than one candidate exists, one is selected on the basis of range and the amount of panning of the camera required.* The camera is then panned and tilted the amount needed to bring the landmark into the center of the field of view, and a picture is taken. The baseboard-tracking routine described previously² is used to find the segments of baseboard in the picture and to fit them with long straight lines.

Exactly what happens next depends on the landmark type. For a door, the long line nearest the center of the picture is selected, and the true image of the landmark is assumed to be the endpoint of the baseboard segment on that line and nearest the center of the picture. An additional check is made to see that the gap from that point to the next segment is long enough to be a passageway. A convex corner viewed from an angle such that only one side is visible is treated as if it were a door. Otherwise, the intersection of long lines nearest the center

* If no landmark is in view, a suitable message is returned together with a suggested vantage point from which a landmark can be seen. This is one of several "error" returns that can be obtained from the program. The program can also be asked to select a specific landmark, or a landmark different from the ones previously selected.

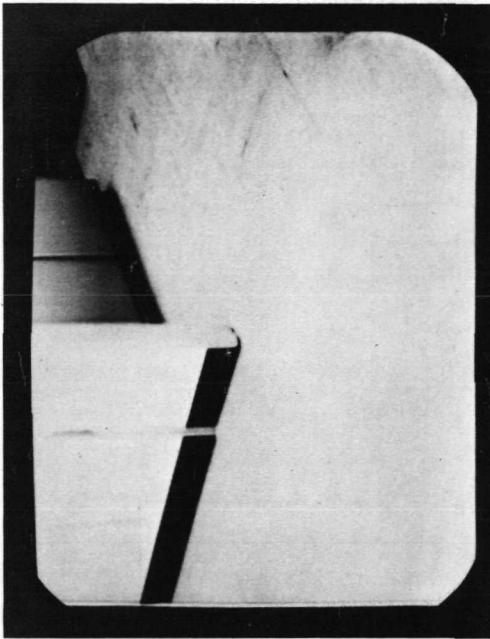
of the picture is assumed to be the true image of the landmark, and a check is made to see that the baseboard segments near this point have the right geometrical configuration. The location of the landmark in the picture gives the information needed to compute corrections for the automaton's position and orientation.

The operation of this program is illustrated in Figure 7. In this experiment, the automaton was approximately 7.5 feet away from a wall along which there were four landmarks, both sides of a doorway, a convex corner, and a concave corner. The pictures in Figure 7 show how closely the panning and tilting brought the landmarks to the center of the pictures. For scenes as clear as these, the program operates very reliably. Presently, we can use this routine to locate the robot with an accuracy of between 5 percent and 10 percent of the range, and to fix its orientation to within 5 degrees. Since the errors are random, the accuracy can be improved further by sighting a second landmark. Further increases in accuracy, if needed, will have to be obtained by improving the tilt and pan mechanism for the camera.

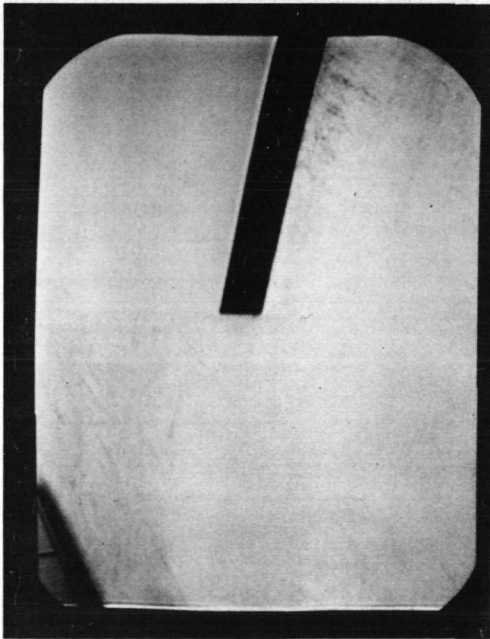


TA-8259-22

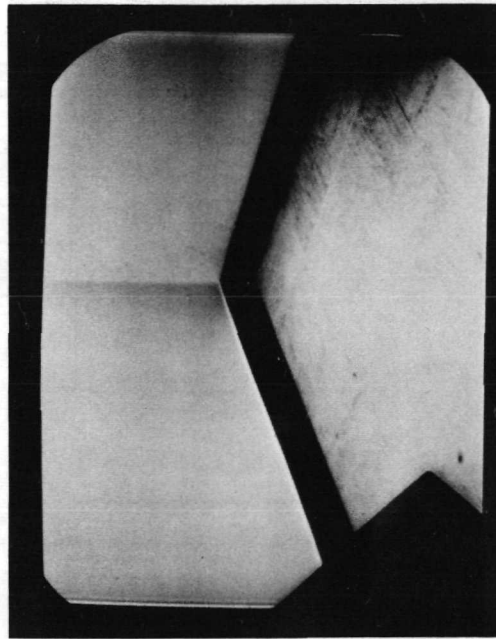
FIGURE 6 BASIC FLOWCHART FOR LANDMARK PROGRAM



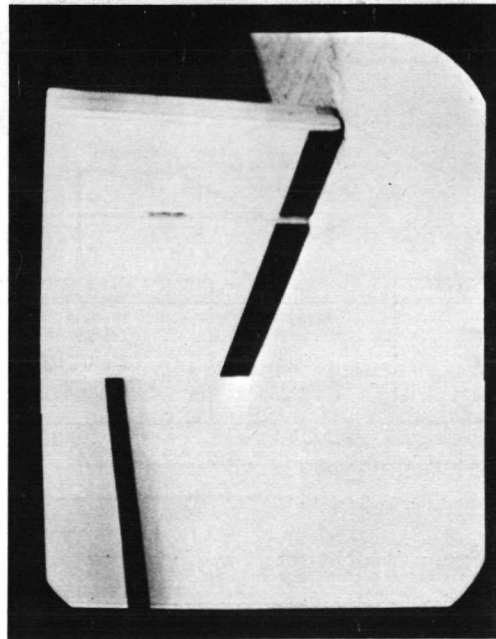
(b) LEFT DOOR



(a) RIGHT DOOR



(d) CONCAVE CORNER



(c) CONVEX CORNER

TA-8259-23

FIGURE 7 LANDMARKS

REFERENCES

1. L. S. Coles et al., "Applications of Intelligent Automata to Reconnaissance," Final Report, Contract F30602-69-C-0056, SRI Project 7494, Stanford Research Institute, Menlo Park, California (November 1969).
2. L. J. Chaitin et al., "Research and Applications--Artificial Intelligence," Interim Scientific Report, Contract NAS12-2221, SRI Project 8259, Stanford Research Institute, Menlo Park, California (April 1970).
3. C. R. Brice and C. L. Fennema, "Scene Analysis Using Regions," SRI Artificial Intelligence Group Technical Note 17, Stanford Research Institute, Menlo Park, California (April 1970).
4. R. O. Duda and P. E. Hart, "Experiments in Scene Analysis," SRI Artificial Intelligence Group Technical Note 20, Stanford Research Institute, Menlo Park, California (January 1970).
5. R. O. Duda and P. E. Hart, "A Generalized Hough Transformation for Detecting Lines in Pictures," SRI Artificial Intelligence Group Technical Note 36, Stanford Research Institute, Menlo Park, California (July 1970).
6. A. Guzman, "Decomposition of a Visual Scene Into Three-Dimensional Bodies," Proc. FJCC, pp. 291-304 (December 1968).
7. P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. Sys. Sci. Cyb., Vol. SSC-4, pp. 100-107 (July 1968).
8. P. E. Hart and R. O. Duda, "Perspective Transformations," SRI Artificial Intelligence Group Technical Note 3, Stanford Research Institute, Menlo Park, California (February 1969).

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Stanford Research Institute 333 Ravenswood Avenue Menlo Park, California 94025		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE RESEARCH AND APPLICATIONS--ARTIFICIAL INTELLIGENCE			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Final Report 7 October 1969 to 7 October 1970			
5. AUTHOR(S) (First name, middle initial, last name) Bertram Raphael			
6. REPORT DATE November 1970	7a. TOTAL NO. OF PAGES 182	7b. NO. OF REFS 30	
8a. CONTRACT OR GRANT NO. NAS 12-2221	9a. ORIGINATOR'S REPORT NUMBER(S) SRI Project No. 8259		
b. PROJECT NO.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ARPA Order 1058 Amendment 1		
c.			
d.			
10. DISTRIBUTION STATEMENT			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY National Aeronautics and Space Administration 600 Independence Ave., S.W. Washington, D.C. 20546	
13. ABSTRACT This is the final report for the most recent year of a continuing program of research in the field of artificial intelligence. This work follows previous projects that resulted in the design, construction, and demonstration of a "first generation" robot system. The work reported here consists of new research aimed at the development of a more sophisticated "second generation" robot. Although the robot vehicle itself will be essentially unchanged, it will be controlled by a completely new computer hardware and software system. In particular, this report contains detailed descriptions of the computer configuration and the bottom-level software design, two new bases for problem-solving systems (called STRIPS and QA4), and new directions in visual scene-analysis techniques.			

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Artificial Intelligence						
Problem Solving						
Pattern Recognition						
Robots						
Theorem Proving						