AD NUMBER: AD0868871

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

This document is subject to special export controls; 1 Nov 1969, and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMIDD), GAFB, NY, 13440.

AUTHORITY

ST-A PER RADC AFSC LTR, 14 OCT 1971

(2)

AD 868871

RADC-TR-69-464
Final Technical Report
November 1969

APPLICATION OF INTELLIGENT AUTOMATA TO RECONNAISSANCE

Contractor: Stanford Research Institute
Contract Number: F30602-69-C-0056
Effective Date of Contract: 6 September 1968
Contract Expiration Date: 7 November 1969
Amount of Contract: $592,156.00
Program Code Number: 8D30

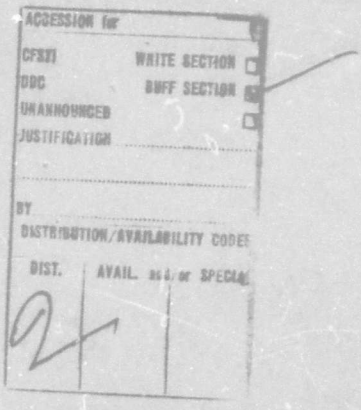
Principal Investigator: Mr. C. A. Rosen
Phone: 415-326-6200

Project Engineer: Miss Patricia M. Langendorf
Phone: 315-330-2621

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1058, Amend. No. 1

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

159



APPLICATION OF INTELLIGENT AUTOMATA TO RECONNAISSANCE

L. Stephen Coles, Bertram Raphael, Richard O. Duda,
C. A. Rosen, T. D. Garvey, R. A. Yates, J. H. Munson

Stanford Research Institute

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Patricia M. Langendorf, RADC (EMIDD), GAFB, NY 13440 under Contract F30602-69-C-0056.

PUBLICATION REVIEW

This technical report has been reviewed and is approved.

Patricia M. Longendyke

RADC Project Engineer

ABSTRACT

A 14-month project of research in the application of techniques of artificial intelligence to the control of a mobile automaton in a realistic environment is described. The main emphasis is on experimentation with a previously-developed system of hardware and software, and on research in several related areas of artificial intelligence where new efforts have been necessary to increase the capabilities of the automaton. Major areas discussed include the use of formal theorem-proving techniques of first-order logic in solving problems for the automaton; symbolic information structures for modeling the automaton's environment; results in visual scene analysis, including a decision-tree approach and the use of regional as well as local analysis; and an outline for the design of a problem-solving system based on higher-order logic.

CONTENTS

ILLUSTRATIONS

BLANK PAGE

SUMMARY

The primary goal of this project has been to investigate techniques in artificial intelligence applied to the control of a mobile automaton in a realistic environment. The basic system of automaton hardware and software was developed under a previous contract.

The principal work performed on this project has consisted of research in problem solving and methods, visual scene analysis, and the design of models of the environment. Section I of this report identifies the place of this project in the context of similar continuing work at Stanford Research Institute.

Progress in improving the current performance of the automaton is described in Section II. One of the major results of the project has been the demonstration that the resolution method for proving theorems in the first-order predicate calculus may be used as a basis for solving automaton problems. Progress has also been made in the development of a computer program that permit a person to communicate with the automaton in a subset of ordinary English. A complex automaton task, which demonstrates the use of the theorem-proving and natural-language abilities, is described in detail.

Section III deals with progress toward a new automaton model. In the present version of the system, the automaton's internal data base (or model of its environment) consists basically of two parts: a hierarchical grid and a relational list structure. We have now decided that the next version will contain a model built up from a single basic storage mechanism, the n-tuple, which is an ordered list of n items of information. The advantages of such a structure are described.

The visual processing system is considered in some detail in Section IV. The system consists of a set of low-level operators for performing local operations on a digitized television picture and an executive program, implemented as a decision tree, for selecting and using the

operators. The problem of extracting three-dimensional information from a picture containing recognizable sections of baseboard with the aid of projective geometry receives special attention. A regional (rather than local) approach to scene analysis is also described.

The long-term problem-solving effort, described in Section V, is involved with the design and development of a general-purpose, formal problem-solving system. This planned system is based upon mechanized theorem-proving in higher-order logic and emphasizes the role of semantic information and flexible control strategies. The work described here has consisted of specifying the necessary features of such a system, and developing the specific languages for the logic and strategy aspects of the system.

Five appendices to the report contain detailed references to various aspects of the research with the automaton system. A sixth lists publications resulting from this contract.

I INTRODUCTION

The primary goal of this program is to investigate techniques in artificial intelligence as applied to the control of a mobile automaton in a realistic environment. The main emphasis has been on experimentation with the system of automaton hardware and software developed under a previous contract, and on research in several related areas of artificial intelligence where new results are necessary to increase the capabilities of the automaton.

The project began in August 1968 as a direct continuation of work performed under a previous contract.[1,2*] Since that time one interim report has been written,[3] and several technical papers have been prepared.[4-9] This report summarizes the work completed to date, referring to relevant sections of the previous publications for background.

No significant changes were made during this project in the basic automaton hardware or underlying software systems as previously described.[1] Instead, the work has concentrated on designing higher-level heuristic methods and programs that try to use the system to solve more difficult problems than had previously been attempted. These efforts to stretch the system's capabilities have led to two results:

(1) The discovery of basic theoretical difficulties that require new research in perception and problem solving before any practical automaton system can be built;

(2) The discovery of limitations on our computer hardware configuration, and basic software organization, that restrict the ultimate capabilities of the system.

---

*References are listed at the end of the report.

Therefore, the principal work performed under this project has consisted of fundamental research in several approaches to problem solving and visual scene analysis, and in the design of models of the environment. This report describes the present status of this research.

This program will be continued in the next year under new support. It is expected that the DEC PDP-10 computer now being installed will provide the necessary expanded memory and computational facilities to permit implementation of many of the ideas developed during this past year and described in the following pages.

II SHORT-TERM PROBLEM SOLVING

A. Use of Formal Theorem-Proving Methods

One major result of this project has been the demonstration that the resolution method for proving theorems in the first-order predicate calculus may be used as a basis for solving automaton problems. Green's paper[5] describes precisely how resolution can be applied to such tasks. The first interim report on this project[3] presents a detailed description of how this technique was used to carry out the task of collecting several movable objects.

In order to use this formal approach to problem solving, we have developed an elaborate software system called "QA3.5." This system contains a resolution theorem prover that uses many of the most powerful heuristics known for obtaining efficient operation, a flexible scheme for storing facts in the form of axioms in both short-term and long-term memory, and a variety of features that permit convenient control and monitoring of the system's operation by the programmer and by the user. In addition to problem solving, the QA3.5 system can potentially be used for maintaining a form of the automaton's model of its environment, and for a variety of "question-answering" tasks of interest to the field of artificial intelligence but beyond the scope of the present project. A detailed description of the facilities in QA3.5 is given in Appendix A.

The effectiveness of QA3.5 as a problem solver is based upon its pure theorem-proving abilities. Therefore, we have devoted some effort this year to the study of the theory of resolution theorem proving. The principal results of this study have been the discovery of a unique graphical representation for classes of equivalent resolution derivations, and a potentially important new theorem concerning resolution proof strategies. A paper describing these results will soon be completed.[8]

B. Natural Language Processing

The primary goal of our research in natural language processing has remained the development of a computer program that would permit a human being to communicate easily with the robot in a subset of English suitable for specifying complex problem-solving tasks. A fairly complete description of the progress made in this area is contained in Ref. 6, the paper presented by S. Coles at the first International Joint Conference on Artificial Intelligence entitled, "Talking with a Robot in English." Within this section, therefore, we only report on work that has been completed since that presentation.

A carefully chosen example illustrates all the essential improvements made during this time period. The example to be considered in detail consists of three English sentences typed by the human and one typed by the computer:

(1) If John is a strong man then he can push boxes that are heavy.

(2) John and Jack are strong men.

(3) Could a heavy box have been pushed by John?

(4) Yes, John could have pushed a heavy box.

Sentences (1) and (2) are facts entered by the user. Sentence (3) is a question, while sentence (4) is the answer to the question provided by the computer on the basis of the first two sentences and some elementary principles of action and time incorporated into its axiom memory as part of its universal knowledge of the world.

The first two sentences are translated into axioms about the world and are represented in the first-order predicate calculus as follows:

(1′) $\text{Is}(\text{John},\text{man}) \wedge \text{Is}(\text{John},\text{strong}) \Rightarrow (\forall x,s)\{\text{Can}(\text{John},\text{push},x,s) \wedge \text{Is}(x,\text{box}) \wedge \text{Is}(x,\text{heavy})\}$.

(2′) $\text{Is}(\text{John},\text{strong}) \wedge \text{Is}(\text{John},\text{man}) \wedge \text{Is}(\text{Jack},\text{strong}) \wedge \text{Is}(\text{Jack},\text{man})$.

The third sentence, being interrogative, is translated into predicate calculus as a theorem to be proved.

(3′) $(\exists x,s)\{Can(John,push,x,s) \wedge Is(x,heavy) \wedge Is(x,box) \wedge Time(s,past)\}$.

On additional fact about actions must be available to answer the question, viz.

(5) $(\forall x,y,s)\{(\forall s)Can(x,y,z,s) \Rightarrow (\exists s)\{Can(x,y,z,s) \wedge Time(s,past)\}\}$.

Loosely translated into English (5) says, "Anything that can happen could have happened." Similarly, there are other principles such as "Anything that can happen might happen," or "Anything that can happen might be happening," but these principles were not needed for this exercise. Note that (5) distinctly does <u>not</u> say that "Anything that can happen did happen."

Taken together, (1′),(2′),(3′) and (5) were used by the theorem prover to affirm the existence of x and s in (3′). The circuitous 18-step proof is shown below.

```
SUMMARY  10/29/69 1159:16

1    IS(JØHN,MAN)                                          AXIØM
                                  1  -  AXIØM
2    IS(JØHN,STRØNG)                                       AXIØM
                                  2  -  AXIØM
3    IS(X,HEAVY)  -IS(JØHN,STRØNG)
      -IS(JØHN,MAN)                                        AXIØM
                                  3  -  AXIØM
4    IS(X,HEAVY)  -IS(JØHN,MAN)                            FRØM 2,3
5    IS(X,HEAVY)                                           FRØM 1,4
6    CAN(JØHN,PUSH,X,S)  -IS(JØHN,STRØNG)
      -IS(JØHN,MAN)                                        AXIØM
                                  6  -  AXIØM
7    CAN(JØHN,PUSH,X,S)  -IS(JØHN,MAN)                     FRØM 2,6
8    CAN(JØHN,PUSH,X,S)                                    FRØM 1,7
9    TIME(SK31(Z,Y,X),PAST)  -CAN(X,Y,Z,SK30(Z,Y,X))AXIØM
                                  9  -  AXIØM
10    TIME(SK31(Z,PUSH,JØHN),PAST)                         FRØM 8,9
11    -IS(X,HEAVY)  -CAN(JØHN,PUSH,X,S)
```

```
     -IS(X,BØX)  -TIME(S,PAST)                          NEG ØF THM
12   -IS(X,HEAVY)  -IS(X,BØX)
     -TIME(S,PAST)                                      FRØM 8,11
13   -IS(X,HEAVY)  -IS(X,BØX)                           FRØM 10,12
14   -IS(X,BØX)                                         FRØM 5,13
15   IS(X,BØX)  -IS(JØHN,STRØNG)
     -IS(JØHN,MAN)                                      AXIØM
                              15  -  AXIØM
16   IS(X,BØX)  -IS(JØHN,MAN)                           FRØM 2,15
17   IS(X,BØX)                                          FRØM 1,16
18   CØNTRADICTIØN                                      FRØM 14,17

8 CLAUSES LEFT.
14 CLAUSES GENERATED
13 CLAUSES ENTERED
14 RESØLUTIØNS ØUT ØF 59 TRIES
SUBSUMED 16 TIMES ØUT ØF 76 TRIES
FACTØRED 0 TIMES ØUT ØF 22 TRIES
6 MIN 55 SEC
```

An affirmative reply from the theorem prover then gave rise to the English answer (4) using the generative grammar. Although the question asked in the example shown hardly presents any challenge from a human point of view, the language and inference requirements are rather sophisticated, and we know of no other computer question-answering system with natural language input capable of handling an example of this complexity with any degree of generality.

The example requires significant grammatical capability:

(1) Conditional sentences (If...then...)

(2) Irregular singular/plural transformations (boxes → box,men → man)

(3) Conjunctions (John "and" Jack)

(4) Relative phrases in predicates (that are heavy)

(5) Active/passive voice transformations (the box was pushed by John → John pushed the box)

(6) Interrogative/declarative transformations (Did John push the box? → Q: John pushed the box.)

(7) Anaphoric Reference (He → John)

(8) Tenses (have been → be)

(9) Subjunctive/indicative mood (could → can).

and no known transformational parsing system even represents "deep structures" in a form suitable for inference making in conjunction with general principles about the world.

Our system, however, suffers from many of the same limitations as other comparable attempts at natural language question-answering, viz., that untutored users tend to force the grammar to grow in an unstable manner, whereas skilled users demand a contraction of the grammar to some formal notation highly suited to a narrow application. These skilled users are more than willing to exchange generality and intelligibility for efficiency in typing and speed of response.

This seeming inherent instability in grammars covering only a fragment of natural language, even though tailored to a particular area of application, will probably remain with us for some time. In one view, however, this will only be as long as typewriters rather than voices are the principal modes of communication between humans and computers. Research in speech recognition is being pursued independently and will probably become available on a limited basis for input to computers within the next five years. Stanford University has already demonstrated such a limited capability in their hand-eye project.[10] In our judgment, recognition of natural English input will become much more important when voice-recognition systems become generally available.

C. The Robot and the Box Problem

During the last several years we have been conducting a series of experiments, each one designed to exhibit increasingly sophisticated problem-solving behavior on the part of the robot. Our earlier experiments concerned exploration of unknown territory and navigation algorithms. More recently we succeeded in having the robot collect a number of specified objects into a designated place. (This task is described in detail in Ref. 4.)

Now, after planning and preparation, we have succeeded in demonstrating a still more sophisticated sort of problem solving, viz., the use of tools. It is believed that this is the first known instance of a computer-controlled robot using an elementary tool as an essential step in the solution of a problem.

More specifically, the problem selected, called "The Robot and the Box" problem, can be stated as follows:

> The robot is in a room in which a box is resting on top of a platform. The robot's problem is to push the box off the platform and onto the floor. (Since the robot is on wheels, no matter from which direction the robot approaches the box, he cannot reach it, since the platform is in the way.) In a corner of the room is a ramp. The solution is then for the robot to push the ramp over to the platform, align it properly, roll up the ramp onto the platform where it can easily push the box onto the floor.

Figures 1 and 4-6 actually show the robot executing the various stages of the solution to this task.

1. Significance of the Problem

First we observe that the robot and the box problem is logically equivalent to another classical problem in artificial intelligence referred to as the "Monkey and Bananas" problem, which can be stated briefly as follows:

> A monkey is in a room in which a bunch of bananas are hanging from the ceiling, just out of reach. The monkey's problem, obviously, is to get the bananas. In the corner of the room is a chair. The solution then is for the monkey to push the chair under the bananas, climb up on top of the chair, where it can easily reach for the bananas.

There is an obvious logical correspondence between the two problems, i.e., both are characterized by having <u>one level of indirectness</u> in their solution. That is, both problems require an auxiliary device not apparently needed at the start of the problem, as an essential ingredient in its solution.

The monkey and bananas problem is of interest, since it has been given to real monkeys by psychologists; although it is somewhat difficult for untrained monkeys, it is possible for them to solve it in time. Thus, we have some frame of reference for our computer systems. An even more important reason for interest in this problem by workers in artificial intelligence, however, is that any computer problem-solving system embedding a logical complete deductive component (which we have) that is capable of solving this class of problems characterized by one level of indirectness, could in principle (mathematical induction) handle problems with an arbitrary number of levels or indirectness, subject to the constraints of space and time.* It should be noted in passing that any problem possessing more than a half dozen levels of indirectness challenges even human ingenuity.

In addition to the above reasons, the robot and box problem is useful in that it exercises practically all of the primitive routines and programs developed for the robot thus far.

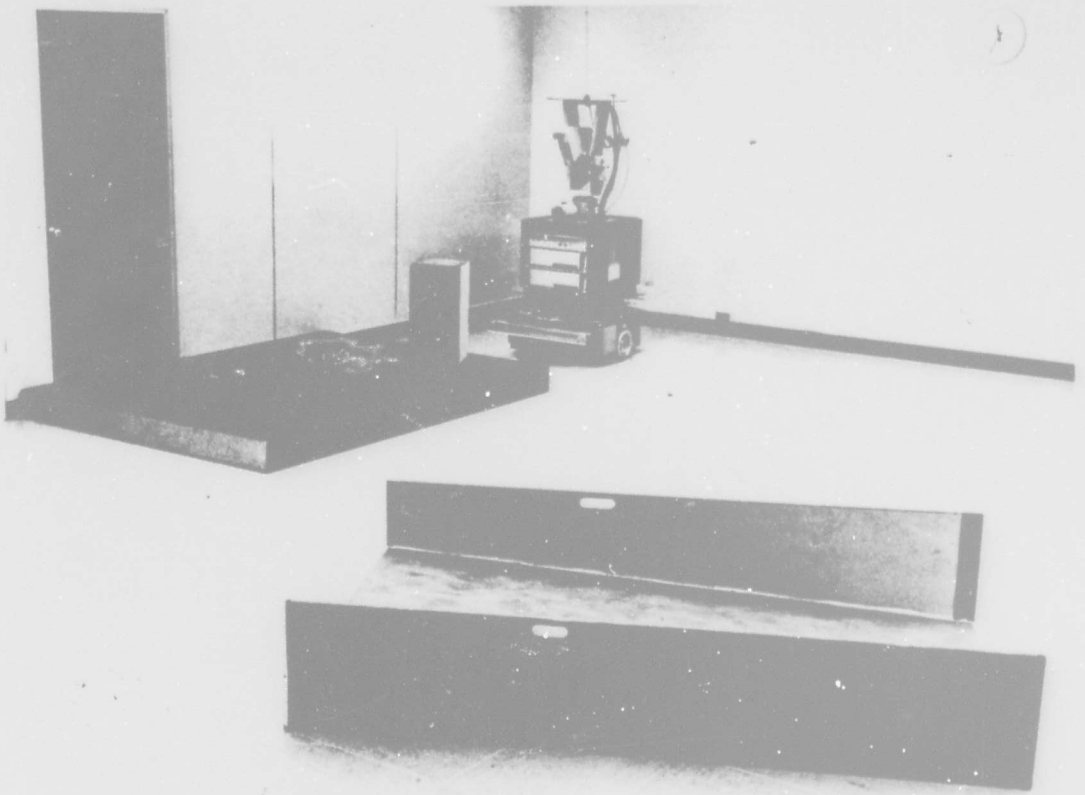
2. Method of Solution

a. Statement of the Problem to the Robot

The initial configuration of the experimental room is as shown in Figure 1. To facilitate matters, we have assumed that the location of the platform in the room and the location of the box on the platform are fixed and known in advance by the robot. Also the robot's current x, y, and θ coordinates and its tilt and pan angles are known to the robot. The first step involves giving the robot some idea of the neighborhood of the ramp by having it roll over to the ramp and bump it. This information is then updated in both the grid and axiom

---

*Although there is a wide spectrum of possible problems ranging in difficulty from the trivial to the impossible; in practice, selecting one that is both nontrivial, yet feasible in a reasonable amount of time, is very difficult indeed.



TA-7494-52

FIGURE 1 INITIAL CONFIGURATION

models with appropriate identification. Note that the position and orientation of the ramp in the room are not known by the robot in advance, and the exact location of the ramp may be fixed by the experimenter at will, subject to the constraint of certain boundary conditions e.g., the ramp cannot be so close to a wall that the robot cannot maneuver it into position.

Now the English imperative sentence "Push the box that is on the platform onto the floor" may be typed to the robot. The first problem for the robot in interpreting the command is to verify that it is a well-formed English sentence. The transformational component of the natural-language grammar is called into play, which (after some inspection) recognizes that the sentence is of the form

push ⟨object⟩ onto ⟨place⟩

where ⟨object⟩ refers to the description of an object and ⟨place⟩ refers to the description of a location. The object and location descriptions are then parsed by the phrase-structure component and the syntactic correctness of the sentence is established.

The value of the initial transformational analysis is seen by considering the possibility of omitting it in favor of a purely phrase-structure analysis. In such a case, the grammar without further semantic information must label the sentence syntactically ambiguous. It cannot distinguish this sentence from the sentence "Push the box that is on the platform near the door," which has an identical surface structure and where the ambiguity is more manifest. Did the speaker intend the robot to push the box so that its final location is near the door or did he intend me to distinguish the box that is on the platform near the door from the box that is on the platform in another part of the room? Only clarification by the speaker or semantic information from the environment can resolve this ambiguity. Obviously there are many ways to proceed. The transformational component merely serves in this case to circumvent this sort of problem where the preposition "onto" makes the correct interpretation obvious.

Concurrently with the syntactic recognition, a deep structure corresponding to the meaning of the command was constructed. The predicate calculus translation of the English sentence constructed by the semantic component is

$$C: (\exists s_i, s_f, x, y, z)\{On(x,y,s_f) \wedge Is(x,box) \wedge On(x,z,s_i) \wedge Is(z,platform) \wedge Is(y,floor) \wedge By(r,push)\} .$$

Loosely paraphrased back into English, the predicate calculus is asking: "Does there exist a final state in which the box initially located on the platform can wind up on the floor, and that this be accomplished by having the robot push? If the answer is yes, then tell the speaker so and do it. Otherwise, tell him why not."

b. Feasibility of Solving the Problem

The next step for the robot, having achieved a formulation of the problem in its own terms, is to figure out how to solve it, given what it knows about its current environment and what it knows about its own primitive capabilities for manipulating its environment, both expressed in a common axiomatic language. The situation calculus (SC) as briefly described in Appendix B, is used to present the problem. Let $P = \{On(x,y,s), At(x,y,s)\}$

where

On(x,y,s) is a three-place predicate denoting that object x is on object y in state s, and

At(x,y,s) is a three-place predicate denoting that object x is at a location adjoining object y in state s.

Further, let $\mathfrak{F} = \{Push(x,y,z,s), Rollup(x,y,s), Move(x,y,z,s)\}$

where

Push(x,y,z,s) is a four-argument function, which maps the state s into a new state in which the agent x has pushed the object y to z;

Rollup(x,y,s) is a three-argument function, which maps the state s into a new state in which the agent x has rolled up the object y; and

Move(x,y,z,s) is a four-argument function, which maps the state s into a new state in which the agent x has moved the object y to a location adjoining the object z.

Finally, let G be given by A1,...,A14 given below:

A1: Is(f,floor)

A2: Is(p,platform)

A3: Is(b,box)

A4: Is(w,ramp)

A5: $On(r,f,s_i)$

A6: $On(b,p,s_i)$

A7: By(r,push)

A8: By(r,move)

A9: By(r,rollup)

A10: $(\forall s)\ [On(p,f,s) \wedge On(w,f,s)]$

A11: $(\forall s)\ [On(r,p,s) \wedge On(b,p,s) \Rightarrow On(b,f,push(r,b,f,s))]$

A12: $(\forall s)\ [At(w,p,s) \wedge On(r,f,s) \Rightarrow On(r,p,rollup(r,w,s))]$

A13: $(\forall s)\ [On(r,f,s) \wedge On(w,f,s) \wedge On(p,f,s) \Rightarrow At(w,p,move(r,w,p,s))]$

A14: $(\forall s)\ [On(r,f,s) \Rightarrow On(r,f,move(r,w,p,s))]$

A15: $(\forall s)\ [On(b,p,s) \Rightarrow On(b,p,rollup(r,w,move(r,w,p,s)))]$

where $\Omega = \{f,p,b,w,r\}$.

Now, with the above formulation in the SC, we may pose such problems as:

P1: $(\exists s_f)\ [At(w,p,s_f)]$

P2: $(\exists s_f)\ [On(r,p,s_f)]$

P3: $(\exists s_f)\ [On(b,f,s_f)]$,

where P3 is a simplified version of the robot and the box problem as stated in the preceding section. The solution to P1 is simple: s_f = move(r,w,p,s_i), which is obtained by one application of A5, A10, and A13 instantiating s_i for s. Similarly, with a somewhat greater amount of effort the solution to P2 becomes: s_f = rollup(r,w,move(r,w,p,s_i)). The solution to the robot and the box problem under the simplified formulation P3 is then seen to be s_f = push(r,b,rollup(r,w,move(r,w,p,s_i))). The solution to the complete formulation by the theorem prover is then

$$s_i = s_i$$

$$s_f = \text{push}(r,b,\text{rollup}(r,w,\text{move}(r,w,p,s_i)))$$

$$x = b$$

$$y = f$$

$$z = p$$

Note that while Axioms A14 and A15 appear to be superfluous at first glance, they are indeed necessary for the solution. One must explicitly state that the robot's being on the floor is an invariant under Move and the boxes being on the platform is an invariant under Rollup, etc. This apparent difficulty is an instance of a more basic problem characteristic of the SC as a whole that John McCarthy has referred to as the "frame problem." That is, in writing axioms in the SC, one must fully specify not only what each action does do in changing the relevant features of a situation, but also what each action does not undo. Various schemes have been proposed for providing a general solution to the frame problem, but yet no one has achieved a fully satisfactory method that can be implemented within the framework of the first-order predicate calculus.

One should also remark that, although the solution to any given problem may be feasible in principle, other prohibitions on robot behavior that are also entered as axioms by the user may prevent the robot from actually carrying out any plan establishing feasibility.

c. Execution of the Planned Solution

Having established the feasibility of a solution, i.e., discovering from the initial state an obtainable final state that satisfies the requirements of the problem, it remains to execute the changes of state. Why not use the solution to the feasibility stage constructively to guide the execution? This amounts to first telling the user that you are about to begin, unwinding the primitive functions composed in the solution, and evaluating them in reverse order with their appropriate arguments. "Telling the user" amounts to generating the English sentence "I will now push the box on top of the platform onto the floor." The execution of primitive functions with proper arguments entails appropriate calls on LISP routines, which in turn call the two-letter FORTRAN commands that actually drive the vehicle through its motions. (The two-letter commands are defined in Ref. 1.) A typical sequence of two-letter commands corresponding to the Move-Rollup-Push sequence is

XG-1.,YGO.5,GOTU51.,WHTI-5.,REPIRVPU9.1,9.5,4.6,4.4,19.1,

TU-90.,MO6.5,TU60.,MO7.5,XG4.5,YG25.,GOMO2.5,OV3,TU-105.,MO6.,

MO-1.,TU105.,OVO,MO7.,XG14.,YG15.5,GOTU-180.,MO15.,TU-90.,

OV3,MO3.,MO-3.,OVO,TU-90.,MO14.,

The above string of 37 two-letter commands are meaningful only to those highly familiar with robot operations, but they do serve to illustrate the kind of complexity one rapidly encounters when attempting even the most trivial of tasks. For the benefit of those not versed in this intermediate command language, we briefly describe in English what actually does transpire in the robot's mind during execution.

> My first subtask is to execute Move(r,w,p,s_1) or, in other words to move the ramp over to the platform and align it properly. To do this I must first discover where the ramp is. To do this I must first see it. To do this I must first go to a place where, if I looked in the right direction, I might see it. This sets up the subsubtask of computing the coordinates of a desirable vantage point in the room, based on my approximate knowledge of where the ramp is. Next I have the

problem of getting to the vantage point. Can I go directly or will I have to plan a journey around obstacles? Will I be required to travel through unknown territory to get there if I go by an optimal trajectory, and if so, what weight should I give to avoiding this unknown territory? When I get there I will have to turn myself and tilt the television camera to an appropriate angle, then take a picture in. Will I see a ramp? The whole ramp? Nothing but the ramp? Do I need to make a correction for depth perception, because I know my vision is not terribly good even under the best of circumstances? etc., etc.

This sort of reasoning in the form of primitive routines and reflex actions goes on and on until finally the robot successfully takes in a picture of the ramp with a reasonable probability of error (Figure 2). Based on this picture, a model of the experimental room is constructed, which might look as indicated in Figure 3.

The angle α is estimated and a strategy for reducing α to an angle within acceptable tolerances is determined. This amounts to reorienting the ramp with an appropriate number of rotational pushes. Next comes a long translational push (cf. Figure 4) putting the ramp into actual contact with the platform. Now comes the ticklish job of alignment with all its inherent subtleties: getting in the right place to push at the right place by the right amount to cause the high side of the ramp to come up flush against the platform with gap of at most one half inch. To make a long story short--once alignment has been accomplished successfully, the rollup (Figure 5) and the push (Figure 6) can be handled gracefully.

3. Operational Experience

In practice, the robot--as currently configured with software implemented on the SDS 940--is not terribly good at getting up on top of platforms in a reasonable amount of time. The recognition of the English command takes in the neighborhood of 90 seconds, even with a comparatively light load on the time-sharing system of the SDS 940 computer. Proof of the theorem establishing feasibility involves only 29 steps with well-tailored axioms, yet still takes 20 minutes on the

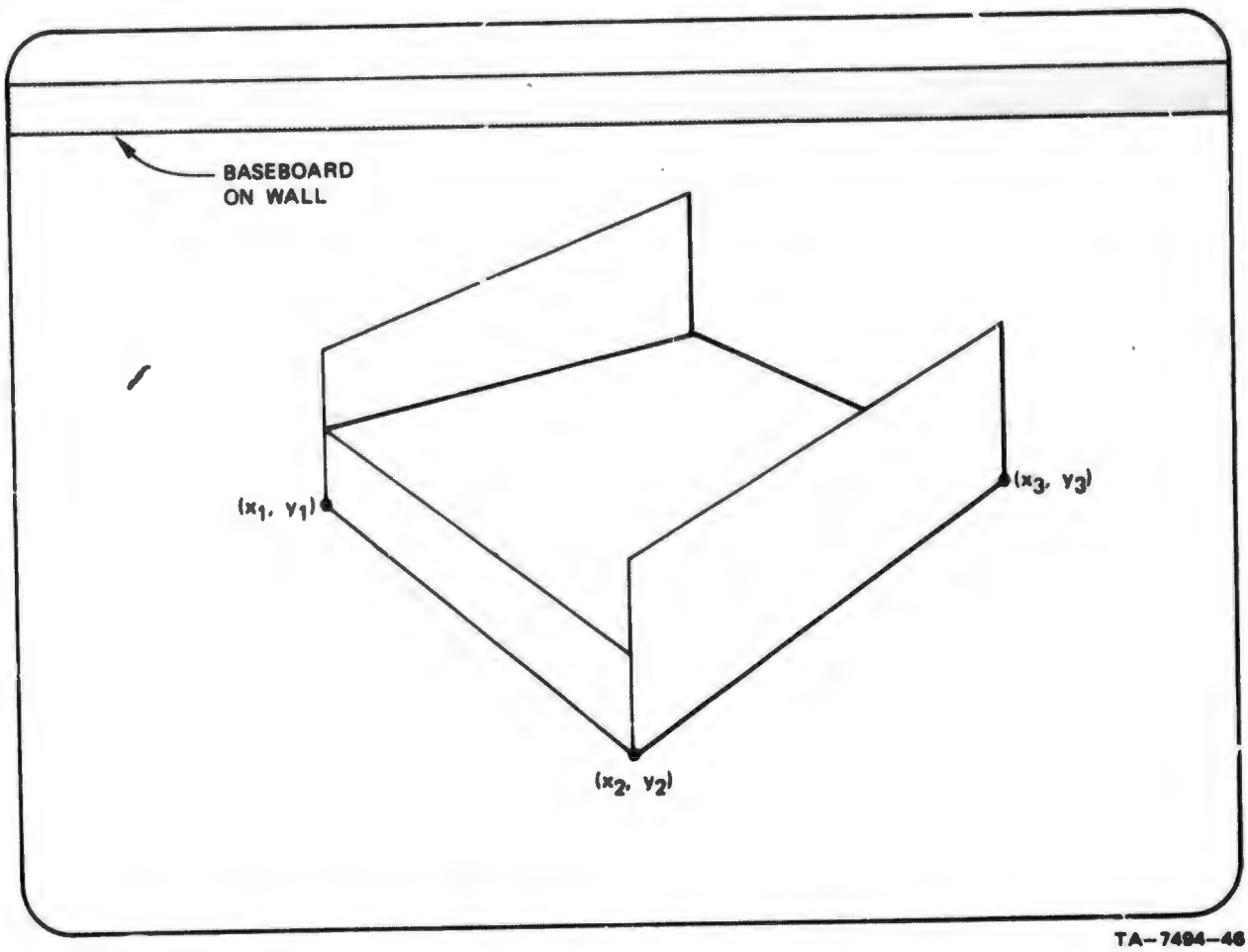


FIGURE 2 TV MONITOR IMAGE OF RAMP

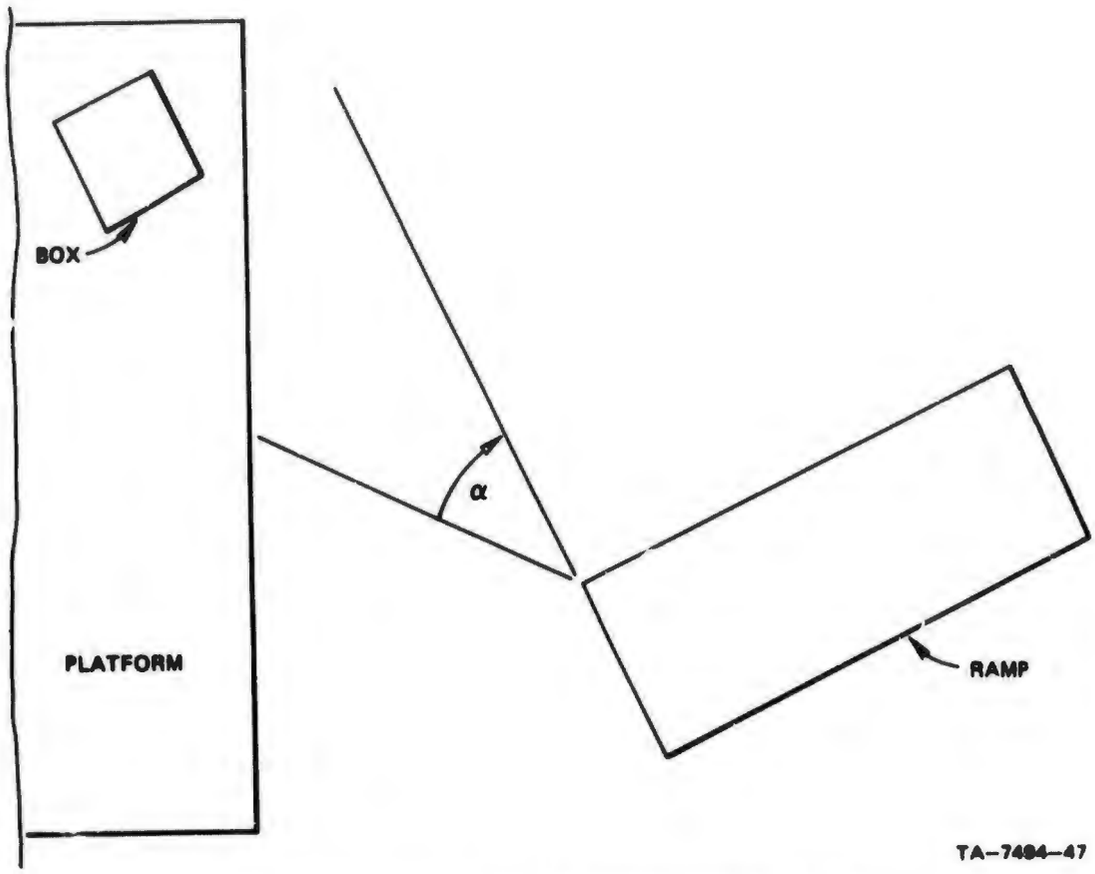
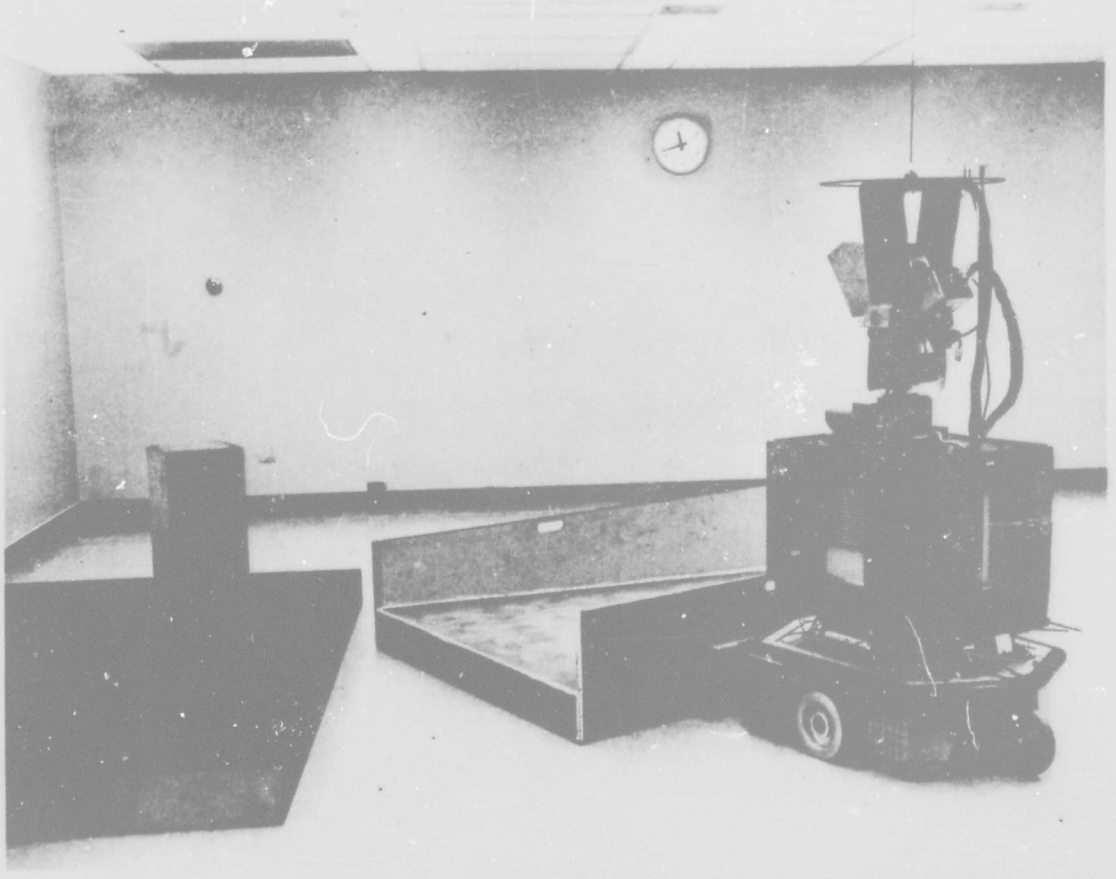
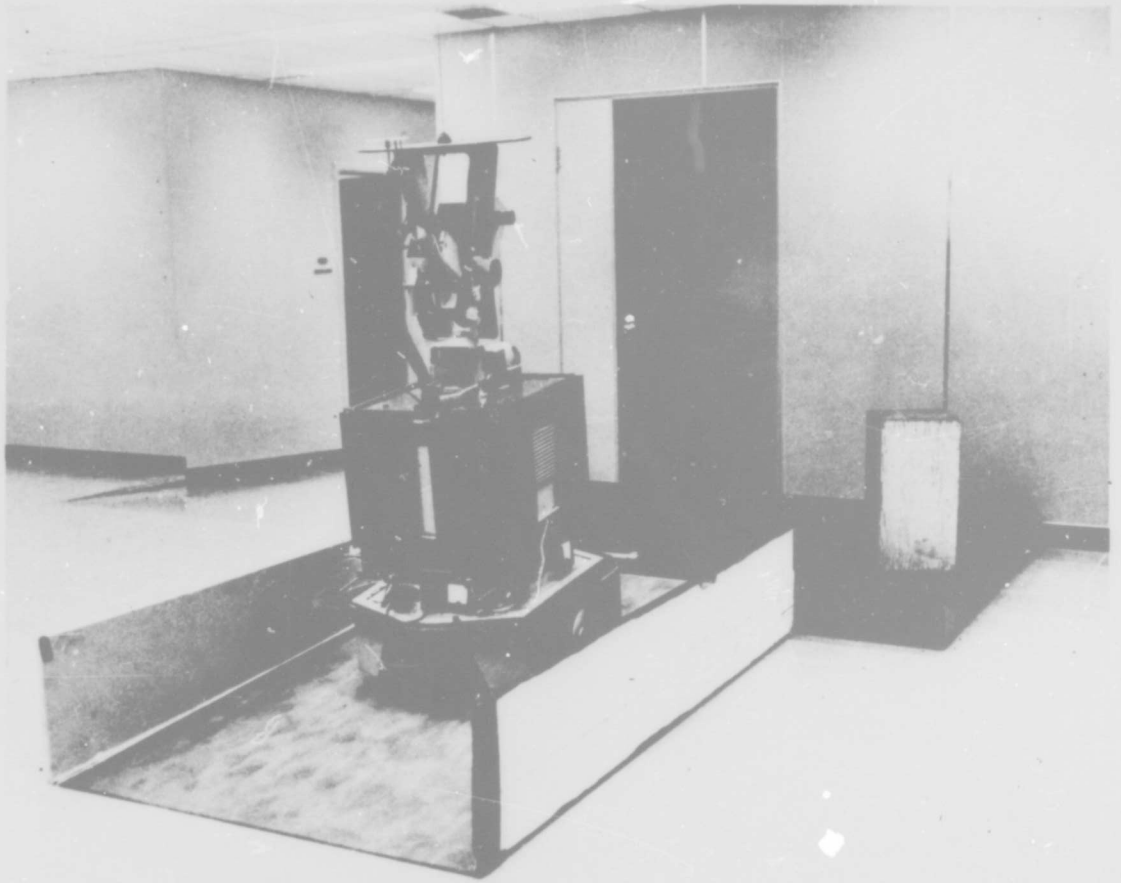


FIGURE 3 PLAN VIEW OF EXPERIMENTAL ROOM



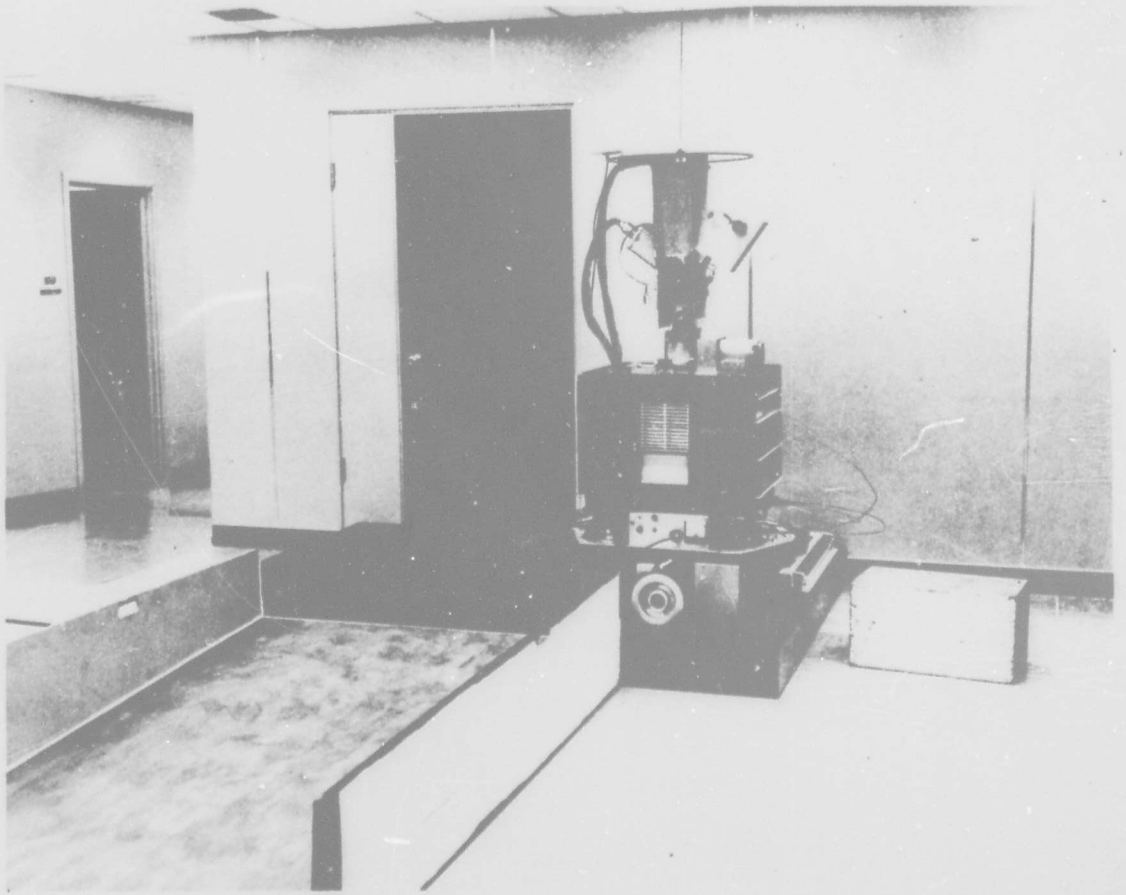
TA-7494-53

FIGURE 4 TRANSLATION OF THE RAMP



TA-7494-54

FIGURE 5 ROLLING UP THE RAMP



TA-7494-55

FIGURE 6 GOAL CONFIGURATION

average. Actual picture taking and pushing may add another 15 minutes, so total time from beginning to end for accomplishing this elementary task with this level of generality takes over half an hour of real time, obviously not a system competitive with humans.

Even allowing for patience on the part of user, he must be willing to accept the possibility that on certain occasions the robot will simply fail to accomplish the task in practice, even though in principle its strategy for achieving the goal is correct. The principal source of unreliability in the system is poor vision, although cumulative error in the stepping motors contributes to the possibility of failure. With considerable effort in tuning the dynamic range of the TV camera, adjusting the antenna for best reception of a noisy signal (even the time of day seems crucial, since background radiation from adjoining labs can create disturbing radio interference), and carefully calibrating the software for best depth-perception correction factors and other systematic errors on that day, the vision software can generate reproducible ramp coordinates in the actual room to within about an inch, which is sufficient for accomplishing the overall objective.

4. Difficulties Encountered

In addition to the problems enumerated above of "tuning" the robot hardware, and such other hardships imposed by unreliable robot hardware, tape units, or the departure of the SDS 940 disc file auxiliary memory due to revised computer configuration plans, the only unanticipated theoretical problem encountered was the "frame problem" discussed earlier. This difficulty was effectively bypassed by adding a few more axioms to the system rather than really "solved." Why then did the task actually take so long for completion? One can distinguish at least two other classes of unanticipated difficulty worth of comment: software compatibility and pragmatic difficulties.

The "robot and the box" task was accomplished by building a top-level software system out of the existing programs to operate the robot. Some of these programs were originally designed merely to exercise

various components of the robot hardware; others were meant to be trial, experimental versions of the routines for model maintenance, display, etc., for gaining experience with the system before implementing more polished versions. Nevertheless, software-compatibility problems arose far out of proportion to what was originally expected. Frequently, undocumented programs designed for one purpose had to be expediently modified for another slightly different purpose; the side effects became evident only at a later time. On other occasions, we found reasonably well documented programs with unarticulated assumptions about the use of memory, a commodity that was abundant when the software packages were checked out in isolation, but scarce when various packages were linked together to operate in concert. Marshalling resources and manpower to carry out the linkage of communicating software packages already in existence also proved difficult. Occasionally the author of a particular routine was no longer with the Institute. Literally weeks were spent in articulating software conventions of different subsystems--such as vision and tactics--so that appropriate parameters could be passed between them.

Some of the most trivial inconsistencies in conventions took the most time to locate and patch. For example, one group measured angles in degrees--the other in radians; one group established a coordinate system 90 degrees out of phase with the other, etc., and (perhaps most humiliating), the LISP system was perfectly content to pass the FORTRAN system the number "51.000," whereas the FORTRAN system in certain routines could only accept the number 51 if it were formatted "51.00." This supposedly is an inevitable consequence of different teams of individual software designers working too long in isolation of one another and in the absence of a master specification to guide them. The system software in which the various packages were embedded with its inherent rigidity, practically nullifying the advantages of working on a time-sharing system, contributed its own debugging problems,

further obfuscating the loading and initialization process. For the new PDP-10 system we are designing a complete new software structure (see Section III of this report), which will provide a uniform framework for future robot experiments.

What we have chosen to call pragmatic difficulties added a whole new dimension to the conventional hardware/software debugging process. Three brief examples are given.

(1) Given that the ramp and the platform had already been designed and fabricated by the carpentry shop, with due attention to the proper dimensions and distribution of weight for use by the robot, it was discovered that the sliding friction of the ramp on the floor was too great for the robot to push conveniently. Solution: Go to the hardware store, buy metal furniture glides, bring in a hammer from home, hammer them in. Time: one day.

(2) It has been established empirically that the ramp tends to slip and fall out of alignment during pushing if not pushed directly on its center of gravity. Solution: Attach spring-loaded push bar on the robot. Time: three days.

(3) Edges on the ramp still do not show up clearly for proper identification, even though care was taken to paint adjoining faces with highly contrasting colors. Solution: Bring brushes and paint from home, paint certain edges white, tape black paper computer tape on other edges, add more fluorescent lights to the ceiling fixtures. Time: two days.

Suffice it to say that during the past year a considerable number of expedient solutions were rapidly implemented to effectively bypass purely pragmatic difficulties, some requiring ingenuity beyond the call of the average programming or hardware specialist. A few

of them may be regarded as "cheating" by outside observers, i.e., not abiding by certain tacitly stated ground rules; however, it should be noted that, by virtue of perseverance and outright tenacity, none of the fundamental assumptions about the nature of the task were ever compromised along the way.

In conclusion, we are clearly on the threshold of exhibiting that elusive quality called "intelligence" by the machines we are building and programming at SRI. We have a long way to go before we realize practical applications or are taken seriously by the public in this endeavor. (People generally reserve the word "stupid" for other people or systems that exhibit only a little bit of intelligence.) Nevertheless, the completion of this experiment in our judgment represents a significant stepping stone in the direction of this quest for intelligence.

BLANK PAGE

III PROGRESS TOWARD A NEW AUTOMATON MODEL

In the implementation of the automaton-controlling programs on the SDS-940 computer, an internal data base (or model) was used, which consisted basically of two parts: the grid model and the LISP model.* The grid model is a hierarchical arrangement of cells that spatially subdivide the automaton's environment to an arbitrary degree of precision. Each cell is marked as full, partly full, empty, or unknown; from a full or partly full cell reference may be made to the object occupying that cell. The LISP model (also referred to as the axiom model or property-list model) contains information arranged according to relational structure, as opposed to spatial structure. The LISP model is more or less synonymous with the axiom base of the QA3 problem solver when QA3 is working on a task for the automaton. The LISP model primarily contains information about the names and other attributes of objects in the environment. (In addition to the grid and LISP models, a small amount of information describing the automaton's position, TV camera orientation, and operating status is contained on an *ad hoc* basis within the FORTRAN program subsystem of the automaton program.)

The use of the grid model and LISP model sufficed for the development and experimentation described in this and preceding reports. However, a review of the model structure was occasioned by our forthcoming conversion to the PDP-10 computer, with the attendant need for reprogramming. It was found that there was room for improvement on several counts.

(1) Objects are more naturally and more efficiently represented by a set of object-oriented descriptors (in the case of a rectangle, X, Y, θ, length, and width) than

---

*The grid model is described in the First Interim Report of this project,[3] and in the Third Interim Report of the preceding project.[2] Section II of Ref. 3 touches on the LISP model briefly.

as a collection of cells on an arbitrary grid. In the former representation, only X, Y, and θ need be changed when an object is moved; in the grid model, a considerable amount of bookkeeping may have to be done. The resolution of the grid model as it was implemented was not adequate to specify the alignment of a box for pushing.

(2) The separation of the LISP model from the grid model impeded communication between the two models and the routines that used them--a problem that was exacerbated by the necessity of going through a special mechanism (the Valet) to transfer between LISP and FORTRAN.

(3) The grid model handled walls in the same piecemeal fashion as it handled objects. In fact, walls were treated as objects; yet, some routines assumed that objects were convex. This would have rendered it difficult or impossible to deal with a wall forming an enclosure such as a small room.

(4) No regular mechanism was provided for the storage and maintenance of information other than that assigned to either the grid model or the LISP model, such as the position of the automaton.

With this background, a reformulation of the model was planned to go hand in hand with the reprogramming of the software system for the PDP-10. While many of the details of the new model have yet to be specified, the major decisions shaping it can be listed.

The model will be the focal point for intercommunication among subsystems of the automaton software complex (the problem-solver, the executive monitor, vision routines, low-level operating routines). Although each subsystem will undoubtedly use storage structures other than the model in its own workings, when it generates information that is to

be accessible to divergent parts of the overall structure it should keep this information in the model.

By and large, the model will refer to the current state of the automaton and its environment.

The basic storage mechanism in the model will be the n-tuple, an ordered list of n items of information. Since the model will be created within the LISP programming system, the elements of n-tuples will in general be the various constructs of LISP: atoms, numbers, lists, arrays, and arbitrary s-expressions. For example, the fact that the automaton is currently at position (x,y), where x and y are numbers, might be represented by any of the following LISP forms:

```
(CURRENT XROBOT x) and (CURRENT YROBOT y)

(POSITION ROBOT x y)

(POSITION ROBOT (x.y))

(ROBOT (x y))
      :
      :
```

It can be seen that storing information by means of n-tuples of variable length affords almost unlimited flexibility. Which form is best to use for each type of model information must be determined by detailed technical consideration of the programming and operating aspects.

The n-tuple storage system (NSS) has been coded in PDP-10 LISP and is described in Appendix C. A virtue of the NSS (and the prime motivation for keeping information in the form of n-tuples rather than, say, on LISP property lists) is that it allows back-referencing by simulating the action of an associative memory. Thus, if the fact

```
(IN BOX1 ROOM3)
```

is stored, one can ask of the system not only, "Where is BOX1,"

```
(IN BOX1 ?)
```

but also, "What objects are in ROOM3,"

(IN ? ROOM3)

and receive the information that BOX1, and possibly other objects, satisfy this request. How this is done is described in Appendix C.

It is to be noted that not all information stored as n-tuples in the NSS will necessarily belong to the automaton model. Any pieces of information within the system could be kept in the NSS, as desired. In particular, we are just beginning work on the very difficult problem of maintaining information about past states of the environment (memory) and future states (planning).

The position of an object will be kept in the model as suggested above, with X, Y, θ, and a description of size and shape.

Objects such as boxes will be considered entities, as will rooms, doorways, and the robot itself. It is important to distinguish between the name of an entity and the entity itself--if only because the entity may have no name, or multiple names, such as "John's Room" and "Room K2060." Let us use the symbol E to stand for the entity identifier. (In practice, E may be an arbitrary symbol generated on request by the LISP system.) Then the model entries describing a box might look like

(OBJTYPE E BOX)

(NAME E BOX1)

(COLOR E BLUE)

(IN E E_{ROOM})

(XPOS E x)

(YPOS E y)

:

Note that the natural language interface and/or the problem solver must be sensitive to the mapping between entities and their names.

Space will be organized in terms of rectangular space parcels, which are entities. Typically, a room or hallway is a space parcel. A nonrectangular room or very long hallway might, however, be represented as a collection of space parcels. Walls are represented implicitly as the boundaries of space parcels. This method of representation is efficient (it allows a wall to be represented as a single number) and it corresponds to the human conception of a building's layout in terms of areas of space, not walls.

Doorways will be represented explicitly as entities related to walls and possessing two coordinates within a wall. A wall is assumed unbroken (and thus a room is assumed closed) except where a doorway is found in it. Space parcels that make up a single room are represented as having full-width doorways between them. Thus, the most common and least complex occurrences (rectangular rooms and unbroken walls) are represented efficiently by default conditions within the model.

Although no grid model is used to represent the locations of objects and walls, a coarse grid will be used to differentiate between known and unknown (explored and unexplored) regions of a room. The grid will probably partition the room into "tiles" of a few feet square. A known region will be considered empty by default if no object is found there. This grid will be accessible from the NSS, for example,

(GRID E_{ROOM} (an array containing the grid)) .

Certain entries in the model, corresponding to basic information about the state of the automaton, will be updated automatically by the routines that operate the automaton hardware. These entries include the (x,y,θ) of the automaton, the tilt and pan angles of its TV camera, the value of its most recent range reading, the status of the cat whiskers, and various settings within the TV system. Other routines within the PDP-10 can modify any of these entries that are believed to be incorrect. For example, range readings to a nearby wall could be used to correct for cumulative creep error in (x,y,θ) of the automaton.

BLANK PAGE

IV THE VISUAL PROCESSING SYSTEM

A. Introduction

The automaton receives information about its environment through three types of sensors: touch sensors, a range finder, and a television camera. Of these, the television camera has by far the highest data rate, and has the potential of providing the problem-solving system with an accurate, detailed description of the automaton's world. The major task for vision research has been to develop scene-analysis programs that can produce such descriptions from digitized television pictures.

During the first 12 months of the present project, the automaton's world consisted of a large, evenly illuminated, essentially uncluttered exporimental area containing an assortment of objects--boxes and wedges of various sizes. Here the prime task for scene analysis was the isolation, location, and identification of all objects in view. More recently we have begun taking pictures in the corridors, where problems due to uneven illumination, shadows, and reflections cannot be avoided. Here the task of scene analysis expands to include doorway and corridor recognition, simple symbol recognition, and landmark identification.

In this section we describe the different approaches we have taken to solving the scene-analysis problem. Because the nature of the problems to be solved depends so highly on the data, we begin by examining some examples of typical scenes and the effects of quantization. Next we briefly survey past approaches to scene analysis and the difficulties they have encountered in our environment. Finally, we describe our present scene-analysis programs in some detail, including a description of somewhat independent efforts in region analysis and floor-boundary tracking that will probably play more important roles in future scene-analysis programs.

B. Television Data

All of our current scene-analysis programs operate on a single, monochrome, digitized television picture. The picture from a single television frame is quantized into a 120 x 120 array of picture elements, the intensity at each point of which is quantized into one of sixteen levels of gray. The spatial resolution is too crude to allow us to see details such as box seams or floor texture, but is usually adequate for representing the major features of the objects in view. The intensity quantization is less satisfactory, in that more often than not at least one edge of an object cannot be seen merely by observing changes in intensity at that edge.

For example, Figure 7(a) shows a very simple scene displayed on a television monitor. A digitized version of this scene, photographed from a digital display, is shown in Figure 7(b). An alternative representation of the digitized picture is given by the 60 x 60 hexidecimal printout shown in Figure 8, where the characters 0 through F represent gray levels 0 through 15. Transitions in gray level can be spotted by any of various local gradient operators. Figure 9(a) shows the results of applying the Roberts' cross operator[11] and displaying all points at which the gradient was greater than zero. A less noisy picture is shown in Figure 9(b) where only points at which the gradient was greater than one are displayed.

In any of the digitized representations it is clear that the spatial quantization is more than adequate for an object of this size. However, intensity quantization makes it difficult to detect the left interior edge of the box by local operations alone. The change in intensity between the top and the left side of the box is one level of quantization at best. However, one-level changes are never significant by themselves, since gradual changes in reflected light intensity fill the picture with one-level transitions. Thus, while there is ample evidence in the quantized picture to locate and identify the object present as a box, intensity quantization will defeat simple techniques for extracting all of the edge information present.

(a) MONITOR (b) DIGITIZED

TA-7494-32

FIGURE 7 A TELEVISION SCENE

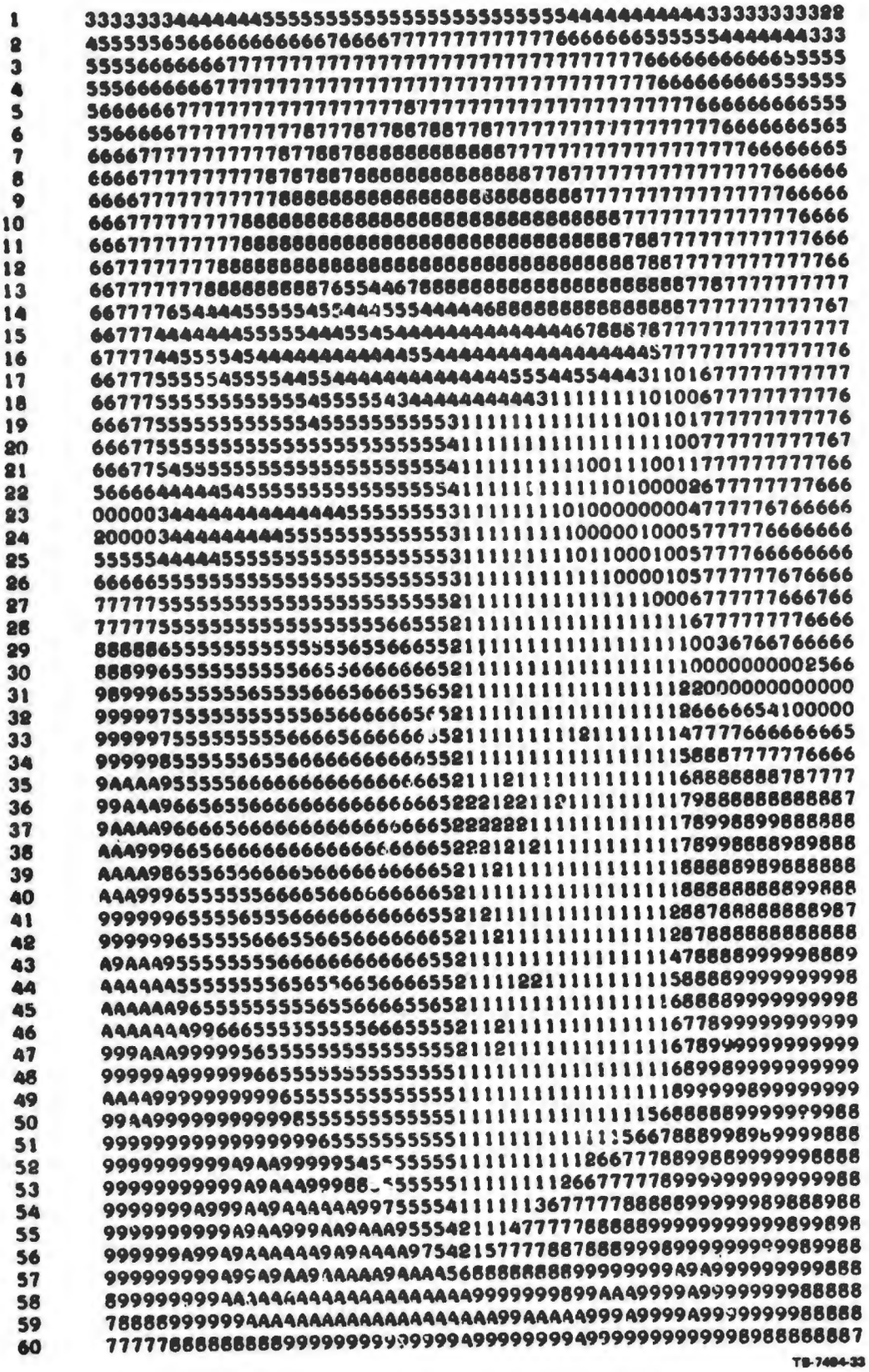
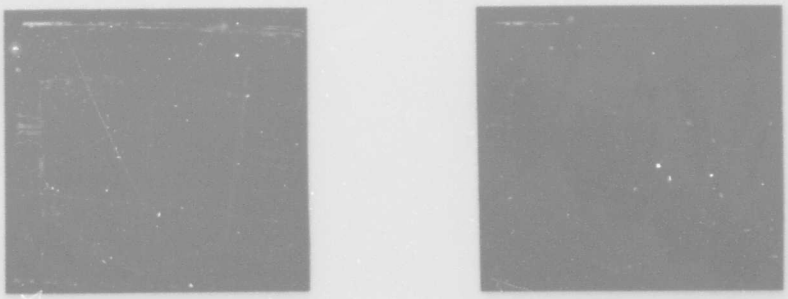


FIGURE 8 PRINTOUT FOR TELEVISION SCENE



(a) GRADIENT > 0 (b) GRADIENT > 1

TA-7494-34

FIGURE 9 GRADIENT PICTURES

These conclusions have been valid for almost all of the pictures we have taken to date. In the course of our work, we have recorded some 130 different scenes, 64 in the experimental area and 66 in the corridors. Figures 10 and 11 show that in either environment the digitized pictures contain valuable information about the automaton's world, but that noise and quantization immediately limit the effectiveness of simple techniques.

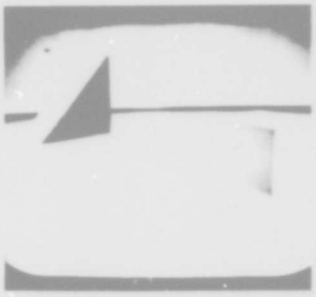
C. Considerations of Scene Analysis

Our initial work in scene analysis was based directly on earlier work by Roberts,[11] and has been described in detail in previous reports and papers.[1-3,12] Basically, it was an attempt to produce an essentially perfect line drawing by a series of local operations on the digitized picture: differentiation, short-line-segment detection, segment grouping, long-line fitting, and long-line connection. If a good line drawing could have been produced, it was assumed that techniques such as those used by Roberts[11] or Guzman[13,14] could have been employed to complete the analysis of the scene.

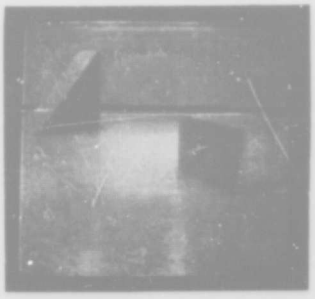
Unfortunately, the goal of producing good line drawings by a series of local operations proved to be unachievable. With so many steps needed to complete the chain, each step subject to error, acceptable results were rarely achieved. The only information that could be reliably obtained by this procedure was the so-called floor boundary, an irregular polygon enclosing area known to be empty floor area. While this is valuable information for the purposes of navigation, it is only a part of the information contained in the digitized picture.

Even though virtually every digitized picture contains areas of difficulty, most pictures also contain strong evidence of one kind or another concerning the presence of objects, together with clues for their identification. For example, many objects contain vertical edges, and a routine that can find strong vertical runs in the gradient picture can provide valuable evidence for the existence of objects.

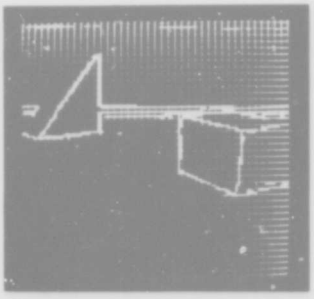
Our current scene-analysis program uses a number of special subroutines to extract evidence from or to apply tests to a picture. These



(a) MONITOR



(b) DIGITIZED

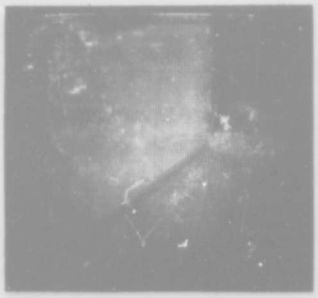


(c) GRADIENT

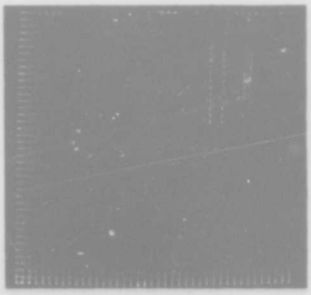
TA-7494-35

FIGURE 10 A SCENE WITH TWO OBJECTS

(a) MONITOR



(b) DIGITIZED



(c) GRADIENT

TA-7494-36

FIGURE 11 A CORRIDOR SCENE

low-level routines operate in an uninformed fashion, in that they perform some operation and return an answer without regard to previous results or the sensibleness of their answers. A higher-level executive program has the responsibility for selecting the operators and piecing together the information they return to build up a coherent description of the scene. We begin our description of the scene analysis program by considering the low-level operators at the disposal of the executive.

D. Low-Level Operators

The executive can call on any of eight different local operators during the course of a scene analysis. These operators may return one or several answers, depending on the operator and the scene. In all cases, associated with every answer is a numerical measure of the operator's confidence that that answer is correct. Typically, confidences measure the strengtn of a response, with 0 corresponding to no response and 100 corresponding to the maximum response.

Masking--One basic test is to establish the existence of an edge, which normally shows up as a quantized line in the gradient picture (see Figures 9, 10, or 11). The mask operator measures the strength of an edge between two specified points by computing the average gradient along the line joining the two points.

Fitting--The fit operator does limited local search to find the best edge response in the vicinity of a line between two specified points. The search is done by systematically placing masks on all lines generated by small perturbations of the specified points, and returning the one having highest confidence.

Verticals--Finding the vertical edges of objects is simplest when the camera is level and the images of the verticals fall in the columns of the picture. To avoid having the overhead fluorescent lights appear in the scene, however, we usually tilt the camera down. This causes the vertical lines in the picture to tilt so that they pass through a common vanishing point on the vertical axis (see Figure 9). The vertical line finder scans the gradient picture for such lines, and returns a list of all lines whose confidence is sufficiently high.

Spurs--Spurs are short segments of lines radiating away from the endpoint of a specified line. The spur finder returns a list of such spurs and their confidences. It operates by computing mask responses for a set of 52 short masks arranged like spokes on a wheel. The table of mask responses as a function of angle is searched for local maxima, using hysteresis smoothing to avoid small irregularities without losing angular resolution. The spur corresponding to the maximum nearest to the specified line is rejected as probably being the line itself, and the routine returns those remaining spurs whose confidence exceeds some threshold.

Directed Spurs--This routine returns that spur at the end of a specified line that comes closest to running in a specified direction. The cosine of the angular error is used to measure confidence.

Following--This routine starts from a specified point and follows the gradient along a specified direction so long as the trace is sufficiently straight. Although this operation is not too reliable, particularly regarding the determination of the proper endpoint, it usually provides a better determination of the angle between a given line and one of its spurs than can be obtained by the spur finder.

Baseboard--This routine tests whether or not a given spur is part of a baseboard. This is currently done by searching the gradient picture for a parallel segment a short distance from the specified spur; we anticipate modifying this routine so that it tests the gray-scale picture instead.

Picture Point--This simple routine merely measures the distance between a specified point and the boundary of the picture. It is usually used to warn the executive that a line of interest may be running off the picture and out of view.

E. The Executive

The executive program selects and applies the operators, and uses their responses to build up a description of the scene. Its basic goal is to isolate and, if possible, to identify the unoccluded objects in

the scene, whether they are either wholly or partially within the field of view. Once an object is found, the lines associated with that object are deleted, and the process is repeated until no more objects can be found.

The executive program has been implemented as a decision tree. Nodes in the tree correspond to an operator to be applied, and branches emanating from a node correspond to the results of that operation. Any path through the decision tree eventually leads to a terminal node corresponding to a description of the location, and possibly the identification of an object in the scene. Repeated passes through the tree produce a list of such information describing the scene.

As the program is being executed, each operator returns its answers together with their associated confidences. The executive program uses these confidence measures to determine whether or not the current line of analysis is reasonable, and to back up and try another path if the analysis becomes implausible. This is done by maintaining a list of open nodes, i.e., nodes that resulted from the application of an operator but were not explored further. Associated with each open node is a confidence, usually computed by averaging the branch confidences along the path from the start node to that open node. The executive always explores the tree by finding the open node having highest confidence, applying the operator called for at that node, deleting that node from the open-node list, and adding to the list the new nodes produced by the operator. Thus, if a wrong turn is taken at some point in exploring the tree, and if the subsequent operators produce meaningless, low-confidence results, then the executive will eventually abandon that approach and start searching another portion of the tree from the highest-confidence starting point.

The details of the scene-analysis program are embodied in the tree itself. While the tree is fairly large and hard to describe in detail, a good idea of how it operates can be gained by following its operation on a simple example. The first step is always to scan the gradient picture and find all vertical lines. Figure 12(a) shows the verticals

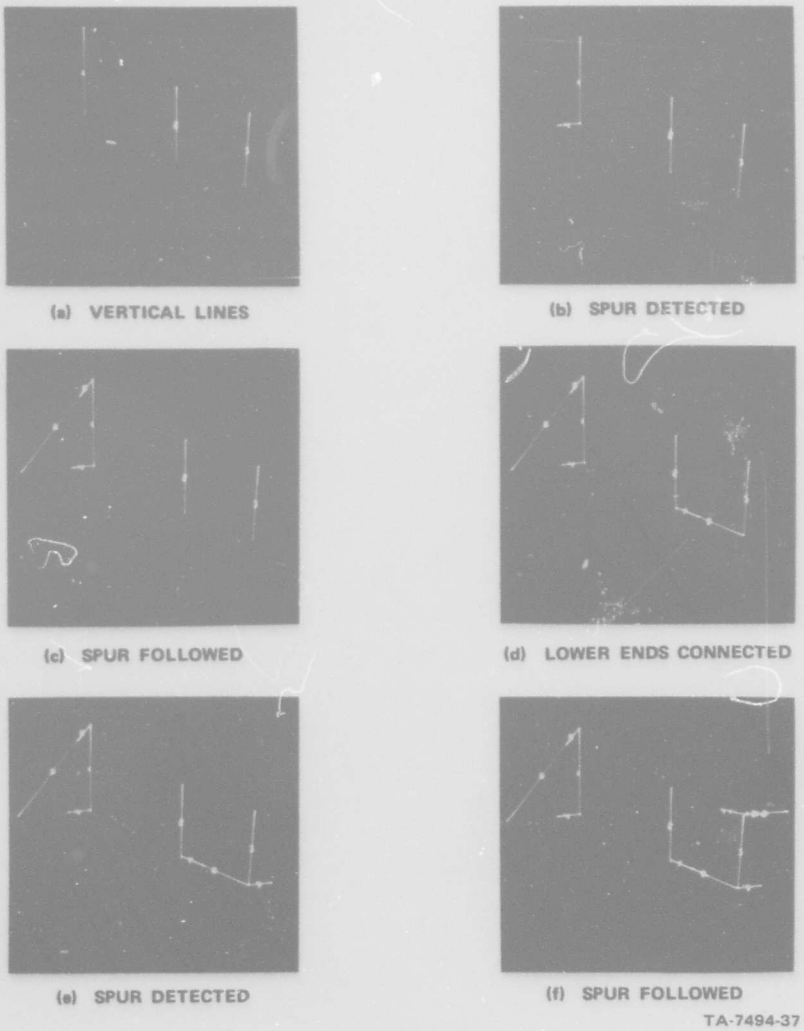


FIGURE 12 STEPS IN SCENE ANALYSIS

found from the gradient picture shown in Figure 10. A check for spurs at the bottom of the strongest vertical discloses, with high confidence, the spur shown in Figure 12(b). Since the baseboard test indicates that this spur is probably not baseboard, the executive is confident that it has found an object, rather than a doorway. An attempt to connect the bottom of this vertical to the bottom of other verticals fails, and attention shifts to the top of the vertical. Here a search for spurs discloses one that can be followed as shown in Figure 12(c), and a test of the angle at the top provides sufficient evidence to state that a wedge has been located.

At this point our present program has reached a terminal node, and the lines associated with the wedge are deleted. The executive then starts over again on the strongest remaining vertical. The spur found at the bottom of this vertical leads to the successful connection of both remaining verticals, as shown in Figure 12(d). Further examination of these lower endpoints discloses the spur shown in Figure 12(e), but since it does not connect to other verticals, attention shifts to upper endpoints. An attempt to follow the upper spur as shown in Figure 12(f) fails to find evidence identifying this object as a wedge, and since other evidence is of low confidence, the executive reaches a terminal node, indicating the existence and location of an unidentified object. At this point all vertical lines are deleted and the finding of objects is completed. All that remains is to find the baseboard and locate the walls, which is done by a special-purpose program to be described shortly.

In its present state, the scene-analysis program can isolate, locate, and (frequently) identify the nonoverlapping boxes and wedges in the experimental area. Objects only partially in view are usually spotted, but are rarely identified. When occlusion occurs, the foreground object can be located and identified, but the program's behavior on the occluded object is unpredictable. To facilitate modifications of the decision tree, the logical description of the tree is coded as a program written in a simple language resembling assembly code for a double-address machine. Basically, the operation codes specify the low-level operator

to be applied, and the operands specify the places to go and actions to take depending on the results of the operation. The burden of keeping open-node lists, backtracking through the tree, handling iteration, and doing other bookkeeping tasks is carried by the decision-tree interpreter described in Appendix D.

F. Some Relations from Projective Geometry

Before we give a description of the routines that find the baseboard and locate the walls, it is convenient to make a digression into projective geometry. The process of taking a picture produces a two-dimensional image from three-dimensional object by a transformation that can be closely approximated by central projection (see Figure 13). Suppose that an object is located at $P = (x,y,z)$ in a fixed Cartesian coordinate system, and let the camera lens center be located at (x_1, y_1, z_1). Let θ be the pan angle and φ be the tilt angle for the optic axis of the camera. Finally, let the object point P be projected into an image point $P' = (x', z')$ found where the ray from P to the lens center pierces a fictitious picture plane located a distance f along the optic axis. If the optic axis pierces the picture plane at (x'_o, z'_o), then the coordinates of the image point are given by:[15]

$$x' = x'_o + f \frac{(x - x_1)\cos\theta + (y - y_1)\sin\theta}{-(x - x_1)\cos\varphi\sin\theta + (y - y_1)\cos\varphi\cos\theta + (z - z_1)\sin\varphi} \quad (1)$$

$$z' = z'_o + f \frac{(x - x_1)\sin\varphi\sin\theta - (y - y_1)\sin\varphi\cos\theta + (z - z_1)\cos\varphi}{-(x - x_1)\cos\varphi\sin\theta + (y - y_1)\cos\varphi\cos\theta + (z - z_1)\sin\varphi} \quad (2)$$

Conversely, corresponding to any image point P' there is a ray in space passing through P' and the lens center. The coordinates of points on this ray are given by:[15]

$$\dot{x} = x_1 + \lambda\left[(x' - x'_o)\cos\theta + (z' - z'_o)\sin\varphi\sin\theta - f\cos\varphi\sin\theta\right] \quad (3)$$

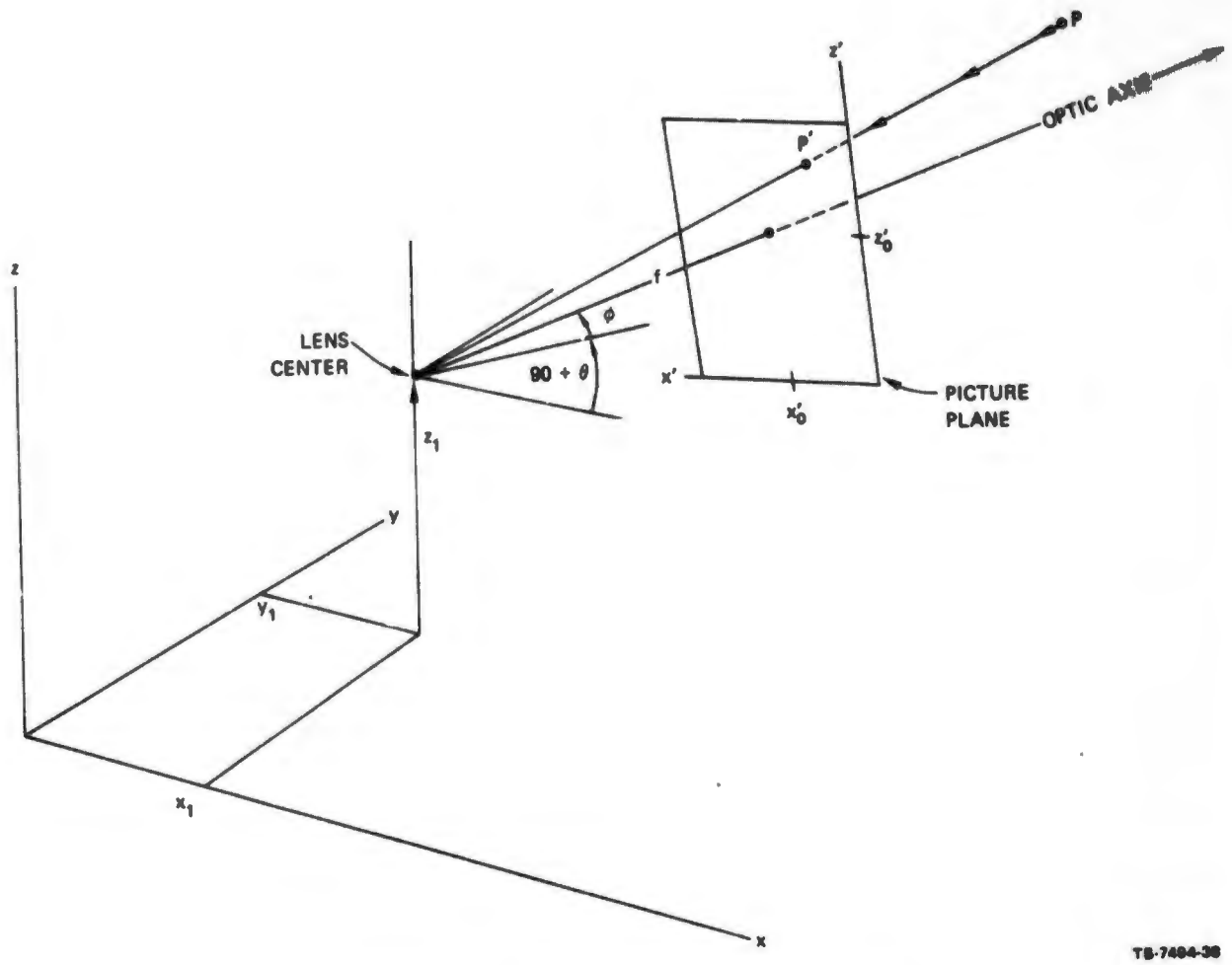


FIGURE 13 GEOMETRY FOR PICTURE TAKING

$$y = y_1 + \lambda \left[\left(x' - x'_o\right) \sin\theta - \left(z' - z'_o\right) \sin\varphi \cos\theta + f \cos\varphi \cos\theta \right] \quad (4)$$

$$z = z_1 + \lambda \left[\left(z' - z'_o\right) \cos\varphi + f \sin\varphi \right] \quad , \quad (5)$$

where λ is a free parameter that fixes a point on the ray. In particular, $\lambda = 0$ corresponds to the lens center, and $\lambda = 1$ corresponds to the image point.

It is shown in Ref. 15 that these equations can be used to derive a number of relations of value to scene analysis. For example, if one of the three coordinates of the object point is known, it can be used to solve for λ and the other two coordinates. In particular, for a point known to be on the floor (the well-known support hypothesis), then $z = 0$, and thus x and y can be found from the image coordinates x' and z' by

$$x = x_1 - \frac{z_1\left[\left(x' - x'_o\right) \cos\theta + \left(z' - z'_o\right) \sin\varphi \sin\theta - f \cos\varphi \sin\theta\right]}{\left(z' - z'_o\right) \cos\varphi + f \sin\varphi} \quad (6)$$

$$y = y_1 - \frac{z_1\left[\left(x' - x'_o\right) \sin\theta - \left(z' - z'_o\right) \sin\varphi \cos\theta + f \cos\varphi \cos\theta\right]}{\left(z' - z'_o\right) \cos\varphi + f \sin\varphi} \quad . \quad (7)$$

In addition to relations of this kind, it is easy to derive a number of necessary conditions on different types of lines. For example, the images of all vertical lines must pass through a common vertical vanishing point located at

$$x' = x'_o \quad (8)$$

$$z' = z'_o + f \cot\varphi \quad . \quad (9)$$

Similarly, the images of all parallel lines in the floor (or in a plane parallel to the floor plane) must pass through the same vanishing point on the horizon line, $z'_h = z'_o - f \tan\varphi$. Furthermore, if the image of

one floor line has the horizon intercept x'_1, the images of all floor lines orthogonal to that line must have the horizon intercept x'_2, where

$$x'_1 \, x'_2 = - \frac{f^2}{\cos^2 \varphi} \quad . \tag{10}$$

Of course, all of these relations are necessary, not sufficient. Thus, if a line in the picture happens to pass through the vertical vanishing point, there is no guarantee that the object line is in fact vertical. However, such tests can certainly be used to eliminate lines that could not possibly be vertical, and each time such a necessary test is passed, the confidence in related conclusions is increased.

In order to use these relations it is necessary to know the quantities x_1, y_1, z_1, φ, and θ, which are geometrical or state parameters of the vehicle, and f, x'_o, and z'_o, which are electro-optical parameters of the television camera. While in theory one or two special cases can be used to establish the camera parameters, we have found it preferable to take more measurements and do least-squares fitting to find the parameter values. For example, if the vehicle is located at the origin and the camera is pointed down the y axis, a floor point at (0,y,0) will image at $x' = x'_o$ and

$$z' = z'_o - f \frac{y \sin \varphi + z_1 \cos \varphi}{y \cos \varphi - z_1 \sin \varphi} \quad . \tag{11}$$

By measuring z' for a series of points $(0,y_k,0)$, we can fit the experimental data for z'_k versus y_k with the ideal curve given by Eq. (11) and find the values of z'_o, f, z_1, and φ that give the best fit. Figure 14 shows a family of such curves, and indicates that if the tilt angle φ is known accurately the support hypothesis can be used to locate points with a range accuracy of about 2 percent.

G. Baseboard Tracking and Wall Location

After the executive has gone as far as it can in locating and identifying objects, the main task remaining is to find the lines where the walls meet the floor, and thus give wall-location information to the

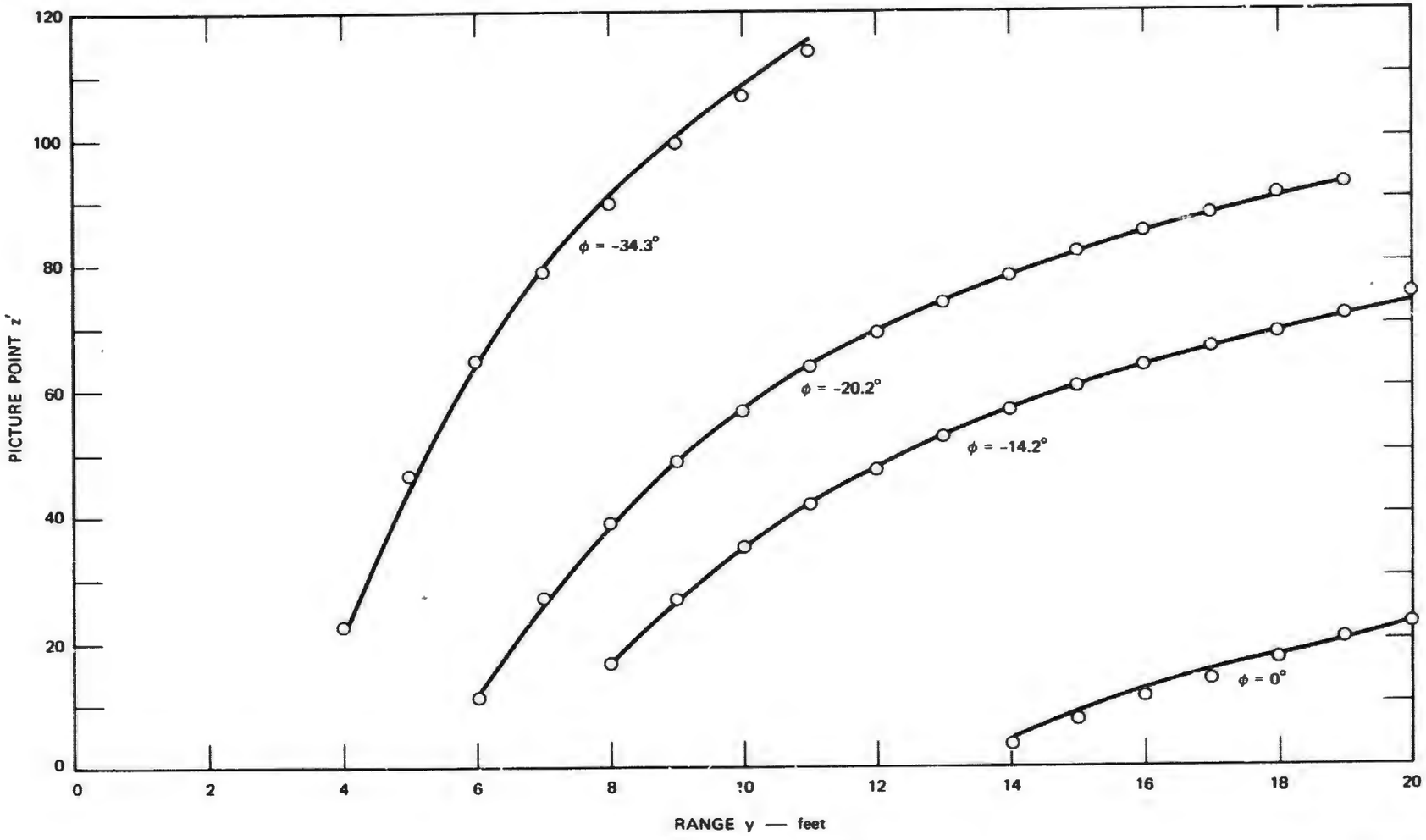


FIGURE 14 CAMERA CALIBRATION

model. This is done by a special program that is composed of two basic parts: a baseboard tracker and a line fitter.

The baseboard tracker scans the gray-scale picture by columns from left to right, searching each column for local minima in intensity that might indicate the presence of a baseboard. Likely candidates are scored according to both darkness and width, the ideal width being a simple function of the distance from the candidate to the horizon. Usually the tracking program finds one or more baseboard segments, with occasional breaks in long segments and occasional false segments. Each segment is represented by a list of $x'z'$ coordinates corresponding to the lower edge of the baseboard. If the coordinates for any segment depart too much from a straight line joining its endpoints, as would happen when the baseboard is tracked around a corner, then the segment is broken in two at the point of maximum departure. After this is done, it is assumed that each segment represents a straight portion of visible baseboard.

The lists of coordinates for the segments are the input to the line-fitting routine. This routine exploits the fact that, in the experimental area, no more than two walls can be seen in any view, and the fact that all walls meet at right angles. Thus, the line fitter attempts to fit the tracking data with either one long line or two lines that meet at right angles. The line fitter effectively operates on a binary picture produced from the coordinates for the segments. The best single long line is found by centering a mask extending across the picture and perturbing it systematically to find the position that captures the most points. The horizon intercept for this line fixes the horizon intercept for the images of all possible orthogonal lines (see Figure 15). Thus, when the line fitter searches for a second long line, it restricts the masks used so that they pass through this other horizon intercept, thereby ensuring that any walls found will meet at right angles. After the highest scoring second line has been accepted or rejected as representing a second wall, Eqs. (6) and (7) are used to compute the location of the walls in world coordinates.

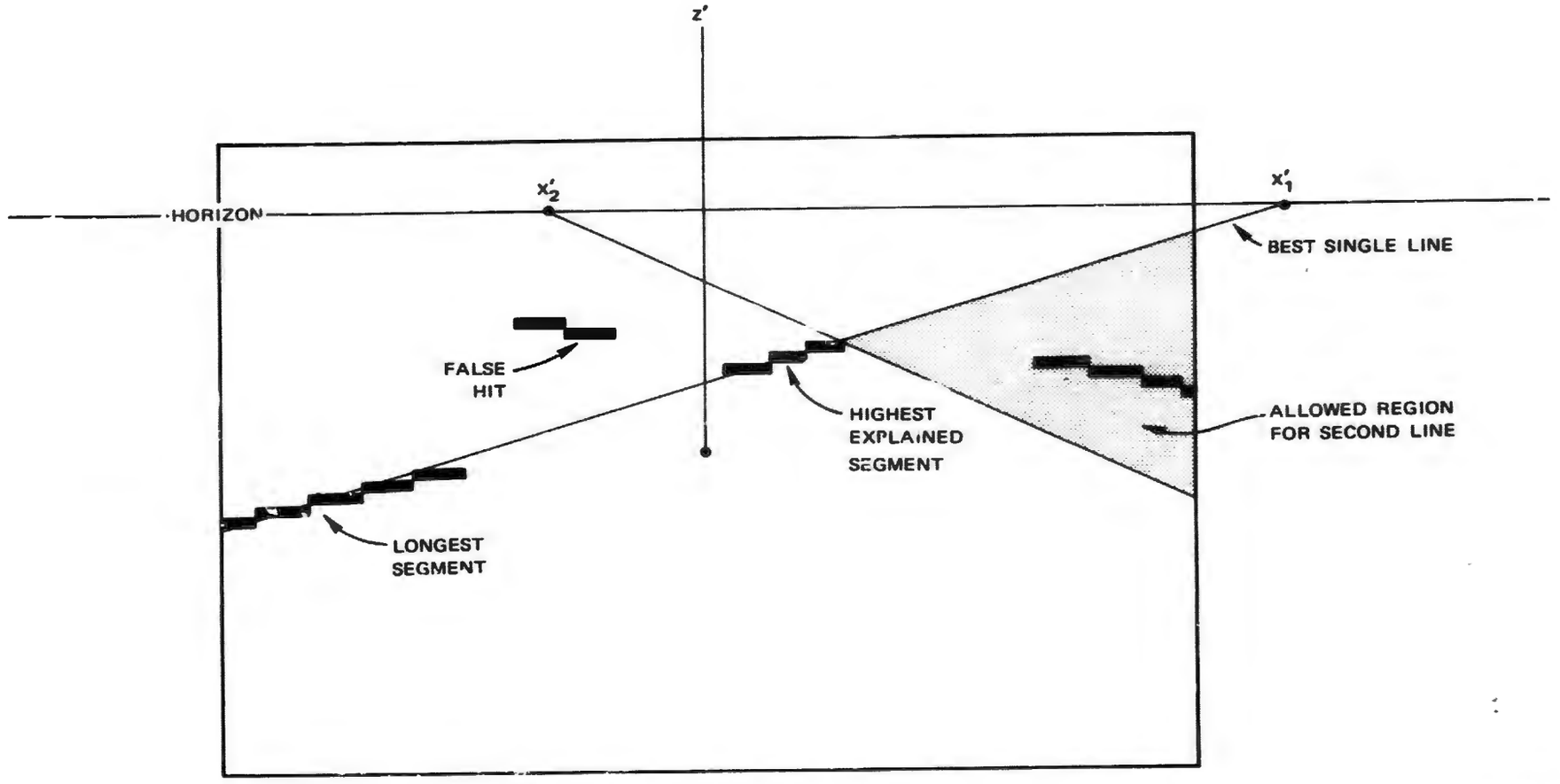


FIGURE 15 GEOMETRY FOR BASEBOARD FITTING

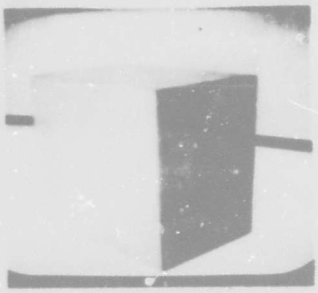
This procedure has proved to be highly reliable at locating walls in our experimental area, and a similar procedure is expected to work well in the corridors. Figure 16 illustrates its operation on the simple scene of Figure 7. The two segments of baseboard seen in both the original image [Figure 16(a)] and the quantized image [Figure 16(b)] are easily tracked [Figure 16(c)]. Neither track is broken [Figure 16(d)], and single long line fits the records well [Figure 16(e)]. Figure 16(f) shows a plan view, where the vehicle, denoted by R, is located at x_1 = 16 ft, y_1 = 0 ft. This wall information is the output of the program, and can be entered directly into the automaton's model of the world.

Figure 17 shows a somewhat more complicated situation in which tracking is a little more difficult, and part of one track goes around a corner. This track is broken correctly [Figure 17(d)], and the long-line fit yields two long lines which project orthogonally in the plan view, as they should [Figure 17(f)]. Finally, Figure 18 shows a corridor scene in which lack of contrast results in a number of breaks in the track and a number of false hits [Figure 17(c)]. A single long line fits the data well, however, and we believe that a modified version of this program will prove to be quite valuable in the corridors.

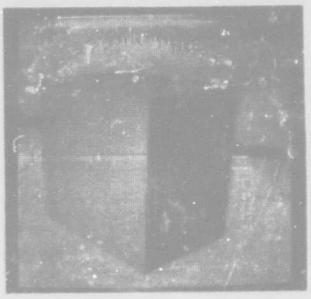
H. Region Analysis

The scene-analysis work just described represents a direct attack on vision problems of immediate importance to the overall automaton project. In addition to this work, we have pursued some independent investigations that may assume a more prominent role in future vision systems. Chief among these has been an investigation of a region-oriented approach to scene analysis. In this section we give an overview of this work, which is described in much greater detail in Refs. 9 and 16.

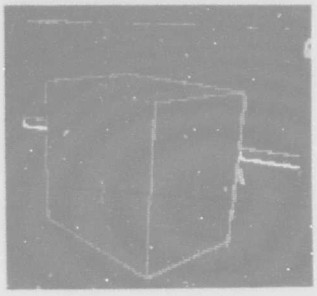
The basic goal of this approach is to partition the picture into meaningful regions--sides of objects, floor area, walls, doors, etc. The basic idea is to start with a large number of small, primitive regions and work toward this goal by operations that can both merge and



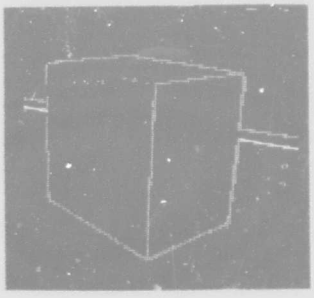
(a) MONITOR



(b) DIGITIZED PICTURE

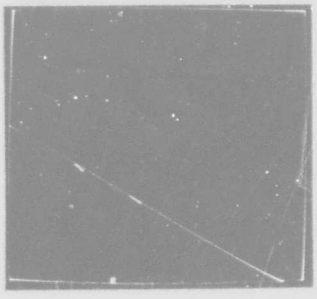


(c) BASEBOARD TRACK



(d) BASEBOARD SEGMENTS

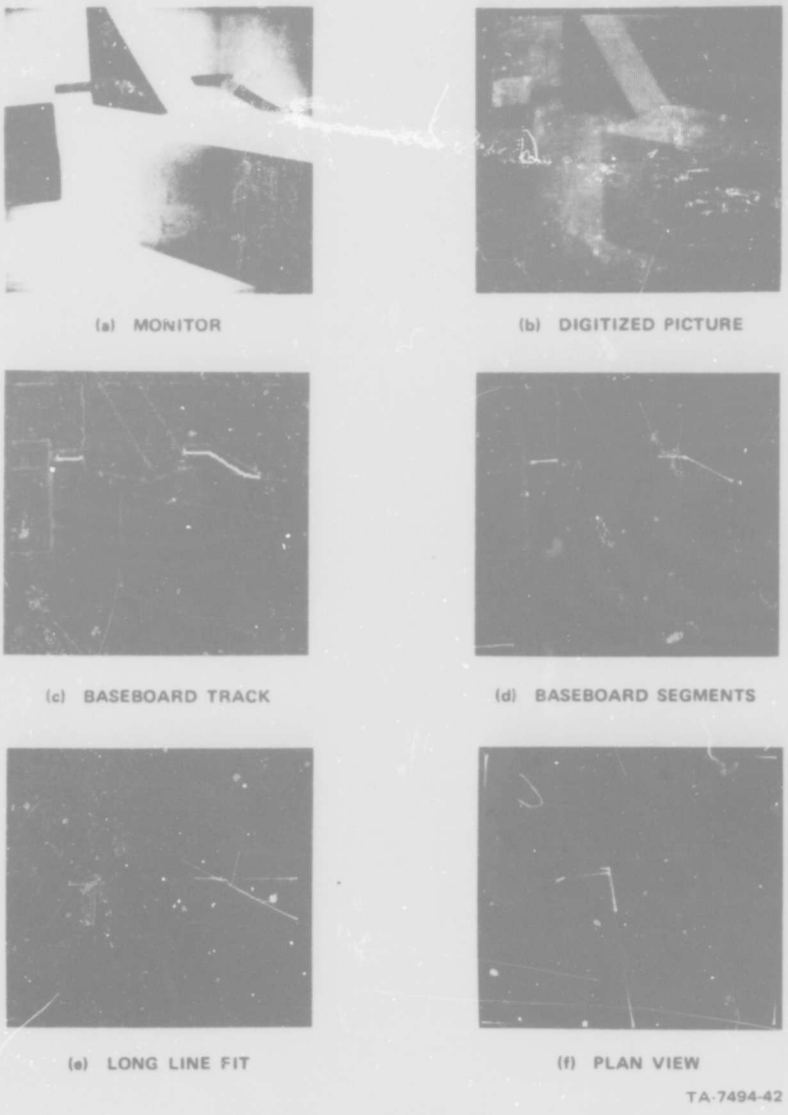
(e) LONG LINE FIT



(f) PLAN VIEW

TA-7494-41

FIGURE 16 BASEBOARD TRACKING



(a) MONITOR

(b) DIGITIZED PICTURE

(c) BASEBOARD TRACK

(d) BASEBOARD SEGMENTS

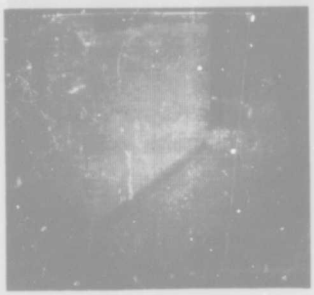
(e) LONG LINE FIT

(f) PLAN VIEW

TA-7494-42

FIGURE 17 BASEBOARD TRACKING

(a) MONITOR

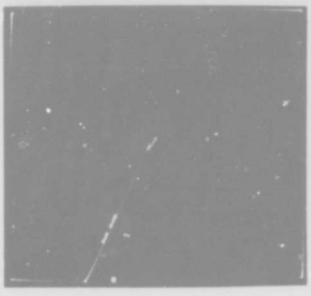


(b) DIGITIZED PICTURE

(c) BASEBOARD TRACK

(d) BASEBOARD SEGMENTS

(e) LONG LINE FIT



(f) PLAN VIEW

TA-7494-43

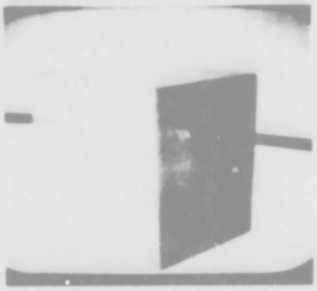
FIGURE 18 BASEBOARD TRACKING

split regions. Most of the work accomplished to date has been on heuristics for merging regions and fitting lines to their boundaries, although plans for using regions in scene analysis have been well developed.

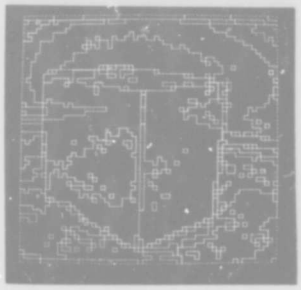
The first step in the process is the partitioning of the digitized image into primitive regions of constant gray-scale value [see Figure 19(a) and 19(b)]. The first heuristic used to merge these primitive regions is based on the fact that the objects in the automaton's world are convex, and when seen unoccluded, their faces form convex regions. Thus, the first heuristic calls for the joining of neighboring regions if they do not differ greatly in intensity and if their common boundary is sufficiently long compared to their perimeters. Repeated application of this heuristic yields the set of merged regions shown in Figure 19(c).

The second heuristic used is based primarily upon intensity differences for neighboring regions. A boundary between two regions is said to be weak if the difference in gray scale values along the boundary is small. Using a weakness measure, this heuristic examines the common boundaries systematically and merges regions along their weakest common boundary [Figure 19(d)]. This heuristic usually simplifies the picture further, yielding a set of regions that can be used as a starting point for more sophisticated scene analysis.

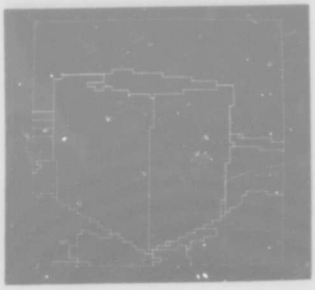
Various line-fitting procedures have been investigated to eliminate the boundary irregularities present even in ideal regions due to quantization. By placing rectangular masks along the boundaries it is possible to fit the irregular curves with straight line segments [Figure 19(e)]. This information can then be added to the region's property list to aid in scene analysis. These routines are under the control of an executive program that tries to form legal figures. By using heuristics like those of Guzman, in simple scenes the executive can combine legal figures to form isolated objects [Figure 19(f)].



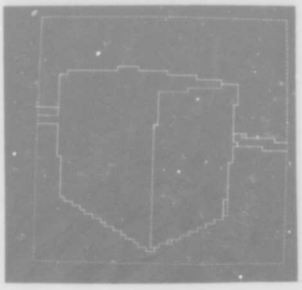
(a) DIGITIZED PICTURE



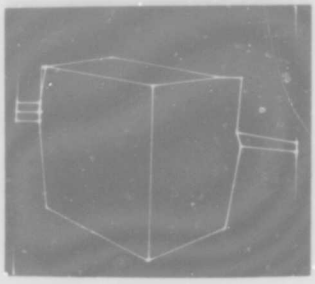
(b) PRIMITIVE REGIONS



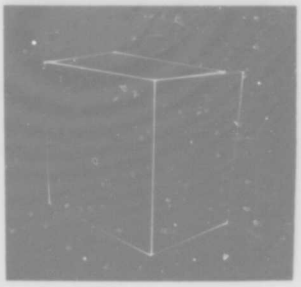
(c) CONVEXITY HEURISTIC



(d) WEAKNESS HEURISTIC



(e) LINE FITTING



(f) OBJECT ISOLATION

TA-7494-44

FIGURE 19 REGION ANALYSIS

The region approach to scene analysis has several appealing features. One advantage is that the same data structure can be used to represent the picture through many stages of processing. Another is that it is easy to change or incorporate new criteria for determining regions. For example, rather than using intensity, one can form regions on the basis of color, range or texture. Finally, by using regions one has a natural and convenient mechanism for tying the scene analysis together with neighboring relations. The parts of an object are neighboring regions, and objects are neighbors of the floor and walls.

I. Status and Future Plans for the Vision System

All of the programs described have been written for and are running on the SDS 940. The decision-tree scene-analysis program and the baseboard tracking program have run successfully on a large number of scenes, although their output has yet to be given to the model and used in problem-solving tasks. This presents no difficulties in principle, however, and it is expected that it will be accomplished shortly after these routines have been converted for use on the PDP-10.

Because the vehicle will not be available for several months during the conversion period, we have recorded 130 different room and corridor scenes for interim work. Several of these are stereo views, and eleven were taken in color. [Experiments on an additional 25 color test pictures are described in Appendix E.] Because the natural color range in the corridors is not large, it appears that stereo and other range information will be more useful than color in that environment, and our initial emphasis will be placed there in new work.

In addition to modifying and extending the present programs, we intend to begin work on new applications for the vision system. Chief among these will be increased use of visual feedback for hall navigation and guiding the automaton through doorways. This includes the use of vision to recognize landmarks and to reorient the vehicle when various errors cause it to lose accurate knowledge of its position. For the first time this will give us a task in which information obtained from

previous pictures and stored in the model will be needed to guide the analysis of the current scene. Finally (although our plans here are not concrete), there is the possibility of beginning a separate effort aimed at developing a different kind of scene-analysis system that interacts more with the problem-solving system and is capable of answering specific questions about a given scene.

V LONG-TERM PROBLEM SOLVING

A. Introduction

The long-term problem solving effort has been involved with the design and development of a general-purpose, formal problem-solving system. The planned system is based upon mechanized theorem proving in higher-order logic and emphasizes the role of semantic information and flexible control strategies. Since a major problem confronting the automaton software effort is to have a general executive system successfully coordinate the robot subsystems (vision, movement operations, models) in the execution of high-level human-initiated commands, such a general-purpose problem-solving system would have immediate applications in this area.

The system proposed here, tentatively called "QA4," represents a significant extension and modification of the ideas incorporated in the QA3.5 system previously described. The formal theorem-proving approach to problem solving presently suffers from major limitations in the following areas.

Semantics--Current theorem-proving systems are essentially pure syntactic procedures. In complex problems, however, significant gains in performance can be made only by focusing attention on the semantics of the problem domain, and by invoking strategies that are highly problem-oriented.

Strategies--The major fault of current theorem provers is their low efficiency--even simple proofs invariably entail a tremendous amount of computation that is irrelevant to the final solution. Efficiency ultimately depends upon the particular strategies and heuristics employed to search the space of possible proofs. New, more efficient strategies are frequently being discovered or proposed. Unfortunately, however, the strategy used by a particular theorem-proving program is generally "frozen" into its code so that newer strategies are extremely difficult, if not impossible, to use.

Level of Logic--No existing program operates in a domain of logic significantly beyond the first-order predicate calculus. However, many interesting problems have a natural, compact formulation in higher-order logic. (These include problems of writing computer programs, and of describing the strategies of the theorem prover itself.) Although such formulations can always be reduced to first-order logic, the reduction is awkward, tedious, and usually obscures the significance of the statements.

The work described below is aimed at overcoming the above limitations.

In our current phase of development, the logic language for QA4 has been largely specified and several proposals for problem-solving strategies, and related operations upon sets, have been under examination. The remainder of this section presents a summary of the proposed system and the work so far accomplished.

B. Capabilities of the Proposed System

The design of the QA4 problem-solving system has been based upon a variety of interrelated features or design goals that we have found necessary in order for the system to achieve the desired power and flexibility. These capabilities are discussed below.

1. Higher Order Logic

The system is embedded in the formal framework of ω-order predicate calculus. This higher-order system will provide great generality and expressive power. By allowing the class of expressions to be part of the semantic domain, the syntax and semantics of any desired portion of the system itself can be expressed in its own language. A recent paper by Robinson [17] develops a foundation for an ω-order logic language and offers a proof procedure for theorems in the language.

2. Expression Evaluation

The evaluation of an expression is the process of discovering that the expression denotes a particular object, or has a particular

object as its value. The concept of the value of an expression is central to higher-order logic, since the semantics of the language as well as all proof procedures are defined in terms of the interpretations (values). Consequently, most of the problem-solving system is devoted to finding the values of expressions. In the case where the expression is computable, that is, expressible in terms of computable constructs, then finding the value reduces to the ordinary process of computation or code evaluation.

One major difference between the proposed QA4 system and the usual "theorem-proving" systems--such as QA3.5--is illustrated by the importance to the new system of expression evaluation, an essentially semantic operation. Methods called "theorem proving" usually denote syntactic logical inference procedures, such as resolution, for which the set of deducible theorems coincides with the set of semantically valid statements. However, in the proposed work we are really only interested in the semantic validity of a statement, and theorem proving is merely a tool that allows us to determine validity. In the higher-order system that we are considering, semantic methods will be used directly to determine validity.

3. Pattern Matching and Transformation

Pattern matching and transformation operations consist of recognizing and naming certain substructures within an expression, and reforming these subparts into new expressions. Since all strategy, evaluation, and state operations need access to the properties of expressions, pattern matching offers a convenient technique for extracting those properties that are dependent upon the syntactic structure of the expression.

4. Set Operations

The proposed system has a set as a fundamental and primitive data type, and sets have a variety of applications throughout the entire system. First, a set is a logical entity and can be easily handled by the higher-order logic constructs; secondly, many sets can have finite

stored representations that would allow logical operations to be explicitly computed by enumerating the set. Finally, by allowing ordered sets we can express many heuristic search procedures in terms of enumerating a set by a given order relation.

5. Representation Changes

These operations will facilitate the use of various representations of information. For example, suppose that, during a complex problem-solving task, three different subprocessors require access to a certain piece of data. Each subprocessor might require the data to be represented in a different form. The three different forms might be functions, predicates, and sets. The representation operators will be able to transform the data into the alternative forms. This facility will also allow experimentation with alternative representations of information and with problems that require finding new representations.

6. Strategy Operations

The purpose of the strategy operations is to create a flexible system capable of change and self-description. The function of a strategy is to specify (schedule) the next operation(s) to be executed or attempted during a process.

A strategy language for describing semantically how a given result is to be achieved is a necessary feature in the system. At present, two proposals have been made for such a language and are undergoing examination and change.

7. Monitor and Control Operations

Monitoring and control operations deal with the flow and processing of information within the system and with respect to the external world. These operations consist of task scheduling, priorities, interrupts, interactiveness, and input/output. The function of the monitor would be as an overriding control unit that relegates the tasks to be performed, accepts and acknowledges interrupts, and acts as the I/O controller. (Information requests and transmission subprocesses

would interface through the monitor.) Since many problem-solving tasks have independent status, it is important to be able to monitor the performances of the individual tasks to determine which subproblems are explored first, and to interrupt control when other subgoals seem more promising. As a consequence, the operation of the monitor interacts strongly with the strategy in operation.

C. Present Status

The current work, in addition to the specification of the features of the QA4 system, has been involved with the development of the logic language and strategy language.

1. The Logic Language

The currently proposed language is essentially an extension of higher-order logic as defined by Robinson.[17] The major additions consist of the addition of the class of expressions to the semantic domain and the addition of set and tuple-valued expressions.

a. The Class of Expressions

The class of expressions is divided into five syntactic categories: identifiers, applications, sets, tuples, and lambda-expressions. An additional rho-expression may be added for certain strategy and set operations.

Identifiers--The set of identifiers is the set of individual symbols of the language such as X, Y, MAX, etc. A certain number of these identifiers--such as AND, OR, =, and EVAL--are taken to have special or primitive meanings in the language. The main property of these special identifiers is that they behave in the same predetermined way under every possible interpretation of the logic language. All other identifiers are called nonspecial.

Applications--An application is an expression of the form (F A), where F is a functional expression (denoting a function) and A is an expression. Intuitively, the semantics of (F A) is the object obtained by applying the function denoted by F to the object denoted by A.

Sets--A set (syntactically) is an expression of the form {E1,...,En}. The object denoted by this expression is the set of E1,...,En.

Tuples--A tuple (syntactically) is an expression of the form ⟨E1,...,En⟩. This expression denotes the ordered n-tuple of objects denoted by the expressions E1,...,En, considered as an element of a set with a product structure, i.e., ⟨E1,...,En⟩ ε S1 x S2 x ... x Sn.

LAMBDA-Expressions--A LAMBDA-expression is an expression of the form (LAMBDA X A), where X is a generic identifier and A is an expression (usually containing X as a free variable). The LAMBDA-expression denotes a function whose value at a point α is the value of the expression A with α substituted for each free occurrence of X in A.

Rho-Expressions--The rho-expression is an expression of the form (RHO X A) and is defined in Section V-C-2-b.

b. Semantics

There are two primitive functions within the language that are used to define the semantics of the language: EVAL and TYPE. The function TYPE is a function from the class of expressions into the class of set-valued expressions, where each expression is assigned a set-valued expression denoting a set to which the expression belongs. For example, the expression (+ 1 2)would be assigned type INTEGER. We have a number of primitive types in the language such as EXPRESSION, TRUTHVALUE, and INTEGER, which denote the set of all logic expressions, the set {true,false} and the set of all integers respectively. If T1,...,Tn are types we can form the new types ⟨T1,...,Tn⟩ and (→⟨T1 T2⟩), which denote the cross product of T1,...,Tn and the set of functions from T1 into T2 respectively. If T is a type, then (SUBSETS T) is the type corresponding to the power set of T(all subsets of T).

Example:

The value of (TYPE "NOT") is (→⟨TRUTHVALUE TRUTHVALUE⟩)

The value of (TYPE "AND") is (→⟨SUBSETS TRUTHVALUE) TRUTHVALUE⟩)

The function EVAL takes the class of expressions into a set (or class) S called the semantic domain or universe of discourse. The value of EVAL applied to an expression is called the denotation or value of the expression. However EVAL is implemented it must obey a number of fundamental rules affecting the syntactic structure and special identifiers defined within the language. We tacitly assume that the semantic domain is large enough to contain the objects denotable by expressions within the language. For example, it must contain the set of expressions, the integers, all power sets of these and so on. Moreover, if T is the type of an expression E, then we always have EVAL:E ϵ EVAL:T. In brief, the function EVAL must coincide with one's intuitive notion of expression evaluation.

c. Special Identifiers in the Language

The special identifiers can be divided into various groups, such as logical connectives, set operators, arithmetic primitives, etc.

(1) Logic Connectives

(a) AND and OR each takes a set S of truthvalues and return a truthvalue. (AND {e1,...,en}) = true only if no ei is false, and (OR{e1,...,en})= true only if some ei is true.

(b) IF takes two truthvalues t1 and t2 and returns false only if t1 = true and t2 = false.

(c) Equality (represented by =) takes a set of objects and returns true only if the set is a singleton, i.e., = {e1,e2,e3} is true, provided that e1,e2,e3 denote the same object. The ≡ equivalence connector in logic is a special case of equality.

(d) NOT takes a truth value and negates it.

(2) Set Operators

(a) The following expressions denote fixed sets.

- TRUTHVALUE (or TV) denotes the set {true,false}
- INTEGERS denotes the set of integers
- EXPRESSION denotes the set of all legal expressions in the language
- INTERVAL ⟨m,n⟩ is the set of integers i with $m \leq i \leq n$
- EMPTY denotes the empty set.

(b) The following operators denote sets formed from other sets

- ⟨S1,...,Sn⟩denotes the cross product S1 × ... × Sn
- UNION {S1,...,Sn} denotes the set S1∪S2∪...∪Sn
- INTERSECTION {S1,...,Sn} denotes the set S1∩...∩Sn
- →⟨S1,S2⟩ denotes the set of functions from S1 into S2
- DIFFERENCE ⟨S1,...,Sn⟩ denotes the set S1-S2-...-Sn
- SUBSETS(S) denotes the set of all subsets of S.

(c) Set Relations

- IN ⟨x,S⟩ is the relation $x \in S$
- INCLUDES ⟨S1,S2⟩ is the relation $S1 \subset S2$
- DISJOINT ⟨S1,...,Sn⟩ states that S1,...,Sn are pairwise disjoint sets.

(d) Subset Operations

- ALL(P) denotes the set of all objects for which P is true
- SOME(n,P) denotes any set of at least n elements for which P is true

- THE(n,P) denotes the unique set of n elements for which P is true
- CHOICE(P) denotes any element for which P is true, else any element
- MAX(S,R), MIN denote the largest or smallest element of S under the order relation R.
- RANGE(f,s) is the set $\{f(x):x \in S\}$
- CLOSURE(S,{f1,...,fn}) is the closure of set S under the operations f1,...,fn.

(3) Tuple Operators

(a) If t is a tuple then (t i) is a meaningful application, denoting the ith element of t

(b) Merge (t1,t2) is the tuple $\langle a1,\ldots,an, bn+1,\ldots,bn+m\rangle$ if $t1 = \langle a1,\ldots,an\rangle$, $t2 = \langle b1,\ldots,bm\rangle$

(c) tmap(f,t) is the tuple $\langle f(a1),\ldots,f(an)\rangle$ where $t = \langle a1,\ldots,an\rangle$

(d) tform$\{\langle \ell_1,e_1\rangle,\ldots,\langle \ell_n,e_n\rangle\}$ is the tuple $\langle e_{j1} \ldots e_{jn}\rangle$ where $\ell_{j1} = 1$, $\ell_{j2} = 2$ ..., $\ell_{jn} = n$.

In addition to these identifiers, others will be defined or modified as the need arises.

d. Quantification

The statements (FORALL X A) and (EXISTS X A) can, following Robinson, be replaced by equivalent expressions using the choice function

(FORALL X A) → ((λ X A)(CHOICE(λ X (NOT A))))

(EXISTS X A) → ((λ X A)(CHOICE(λ X A)))

For certain types of manipulations one or the other of these two equivalent ways of handling the quantified statements would be in use.

2. Strategy and Set Operations

Two candidates have been proposed as a basis for a strategy language; the first is a TRANSFORM statement, and the second is the RHO-expression.

a. The Transform Statement

The generic problem posed to a problem-solving system can be phrased in the following form: Find an object (expression) X satisfying some given syntactic property P, and whose denotation satisfies some semantic property Q. We have the following statement:

(FIND X|P such that Q(X))

where the X|P and Q(X) illustrate the two distinct contexts for x--syntactic and semantic.

Example:

To find the solution to the equation X + 1 = 2 we would say:

(FIND X|numeric such that X + 1 = 2)

The predicate P is a restriction on the representation of the final result. The TRANSFORM statement uses this type of format:

TO TRANSFORM X|P1 into Y|P2 when Q(x,P1) TRY ⟨e1,e2,...,en⟩

which states that given an expression X satisfying a syntactic property P1 and a semantic property Q, to transform X into an expression Y of the form specified by P2 we attempt to evaluate e1,...,en in order. If the execution succeeds then the desired transformation will have taken place. The expressions e_i could themselves be FIND statements or directly computable statements.

Example:

TO TRANSFORM (f A) into Y|P2 when computable(f)

TRY ⟨(FIND $B|_{DOM(f)}$ B = A), Set U = Apply⟨f,B⟩,

(FIND Y|P2 U = Y)⟩ .

This statement says that in order to represent (f A) as an expression satisfying P2 when f has a computable representation, we first find a B semantically equal to A but syntactically within the domain of the representation function. Then apply f to B and finally transform this expression U into an expression satisfying P2.

The complex problem could be solved by searching for applicable TRANSFORM statements and recursively aim toward the reduction to specifically computable forms. In the event of failure by a transform statement, a back-up is made and other applicable statements are tried.

b. The Rho-Statement Expression

The RHO-expression is analogous to the LAMBDA-expression, having the form (RHO X A). The main difference is that whenever an application of the form ((RHO X A)S) is encountered during evaluation (when S is a set) the entire expression is queued and the expression A is executed for each member of the set S. Moreover, if at any future time, an element is added to S, the expression A will be involved in the new element. Since the evaluation of A can assert that elements are members of sets and thereby initiate other RHO forms, the entire process of evaluation can be triggered by a single RHO expression.

RHO expressions could be useful in describing strategies by causing evaluations whenever certain conditions hold in the machine. An example of a particular problem easily phrased and solved by RHO expressions is the heuristic search problem:

(FIND g g ε closure (T,E) s.t. G(g))

which says, find an element g in the closure of a set T under an extension operator E satisfying a goal predicate G.

The implementation of a RHO-expression evaluator on a conventional computer offers certain technical problems, but we believe we see how it can be accomplished. With such a facility, the programmer will be free to plan strategies completely within the framework of set expansion, and be able to combine tree growing and searching techniques with an ease not available in any current programming or problem-solving system.

Appendix A

USER'S GUIDE TO QA3.5 QUESTION-ANSWERING SYSTEM

Appendix A

USER'S GUIDE TO QA3.5 QUESTION-ANSWERING SYSTEM*

1. Introduction

QA3.5 is a question-answering system based on a first-order predicate calculus theorem prover using Robinson's resolution principle. The system is made up of about 240 LISP (written in BBN LISP on the SDS 940) functions, most of which run under QAS, the executive function.

The executive contains provisions for changing the strategies, tracing proofs, unwinding proofs (i.e., printing out only those steps that lead directly to a proof), stepping through proofs by hand, and many different operations on the axiom base.

Many facilities make it easier for the user to perform operations on his data base. This appendix is a compilation of documentation relating to the use of QA3.5. (Portions of it had been previously prepared by C. Green, R. Kling, A. Robinson, and R. Yates.)

The program has proved theorems in group theory, number theory, geometry, and algebra; it has solved "real-time" robot problems; it has been used to draw inferences from data bases containing several hundred axioms. It is hoped that this documentation will allow even more extensive use of QA3.5..

2. Internal Representations

In general, the resolution theorem prover attempts to prove the unsatisfiability of the conjunction of a set of clauses. These clauses originate from the axioms and the negation of the theorem entered by the user.

Data types are terms, literals, and clauses and are defined (and represented) as outlined in the following subsections.

---

*By Thomas D. Garvey.

a. Terms

A term is either a variable (individual symbol) or a function symbol followed by an ordered list of terms. The QA3.5-LISP representation of a term is the following type of S-expression.

(1) Individual symbol--represented by a LISP atom.

(2) Function symbol followed by a list of arguments--represented by a LISP list (f t_1 t_2 ... t_n). The first element is the function symbol (a LISP atom). The remainder of the list is the list of terms (t_1 t_2 ... t_n) representing the arguments.

(3) Constants--a constant is a function symbol followed by the null list of terms. If c is a constant symbol, as a term it would be represented as (c).

Example: The term f(x, g(x,g(y,z)), c) would be represented as

(f x (g x (g y z)) (c))

b. Atoms and Literals

An atom is a predicate letter followed by list of terms. A literal is either an atom or a negation sign followed by an atom.

(1) Predicate letter--predicate letters are represented as positive integers.

(2) Atoms--an atom is represented as a list (p t_1 ... t_n) where p is the predicate letter and where (t_1 ... t_n) is a list of terms.

(3) Literals--a literal of the form $\neg$ A, where A is an atom and is represented as a list (-p t_1 ... t_n), where -p is the negative of the number p representing the predicate letter of the atom.

c. Clauses

A clause is a disjunction of literals. It is represented as the list (HDR ℓ_1 ℓ_2 ... ℓ_n), where ℓ_1 ... ℓ_n are the (representations of the) literals of the clause. HDR is a list containing information relevant to the clause. It has the form HDR = (L level hist answerc props).

(1) L is a list containing the T-support status of the clause plus information pertaining to what clauses have not yet been resolved with c. So L = (T supp units 2-clauses 3-clauses ...).

(a) T-supp is True(T) if c has T-support and False(NIL) otherwise.

(b) units = (un_1 . un_2) where:

- un_1 = (ℓ_i ... ℓ_n) is a pointer into the literals of clause c
- un_2 = (u_j ... u_n END) is a pointer into the list of all unit clauses filed on the array CLAUSEARRAY (described below).

(c) j-clauses = ($c_k c_{k+1} \ldots c_{kn}$ END) is a pointer into the list on CLAUSEARRAY of all clauses of length j. If j-clauses is NIL then, by default, clause c has not yet been resolved against any j-clauses.

(2) Level is the level of the clause defined as 0 for original clauses (axioms and the negation of the theorem) and 1 + max ($\ell(c_1), \ell(c_2)$), for c where c is a resolvent of c_1 and c_2 and where $\ell(c_1)$ is the level of clause c_1.

(3) Hist has several forms, depending on the type of clause.

(a) If c is an axiom, then
HIST = (AXIOM (p_1 ... p_n) (f_1 ... f_k) s)
where s is the original WFF from which the clause c was derived
(p_1 ... p_n) is a list of the distinct predicate letters of s

(f_1 ... f_k) is a list of the distinct function symbols in s.

(b) If c is the negation of the question being asked, then HIST = (NTH).

(c) If c is the resolvent of clauses c_1 and c_2 on literals ℓ_1 of c_1 and ℓ_2 of c_2, then HIST = (RES c_1 ℓ_1 c_2 ℓ_2).

(d) If c is a factor of clause c_1 on literals ℓ_1 and ℓ_2 then HIST = (FAC c_1 ℓ_1 ℓ_2).

(4) Answerc is the answer-clause being built up by the theorem prover.

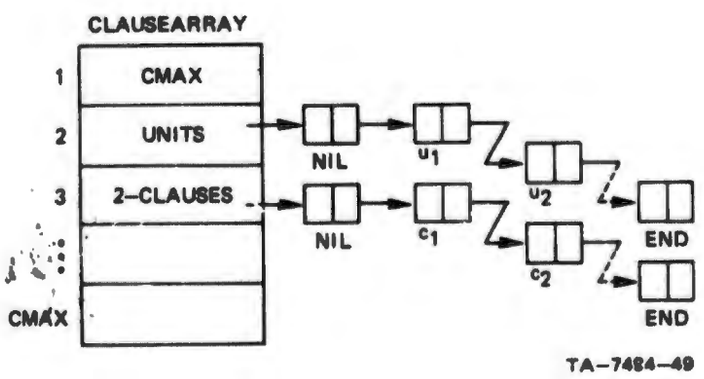
(5) Props is the property list associated with each clause. It has the form (($prop_1$. val_1) ($prop_2$. val_2) ...). This is used to store various properties (such as ROLI, KILL, etc. described below) with each clause.

3. Memory

For greater efficiency, the clause storage in QA3.5 is separated into two stages: CLAUSEARRAY and MEMARRAY.

a. CLAUSEARRAY

CLAUSEARRAY is an array containing the clauses that the theorem prover is currently using. Its structure is as follows:



(1) Clauses are filed on CLAUSEARRAY by length. Clauses of length n (including the header) will go on the list of n-clauses, which is the nth element of CLAUSEARRAY. Hence, unit-clauses have length 2 by this convention and go on the list in the second element of CLAUSEARRAY.

(2) CMAX is an integer that references the first unused location on CLAUSEARRAY. Thus CMAX is greater than the length of any clause on the array. The first element of CLAUSEARRAY contains this maximum CMAX.

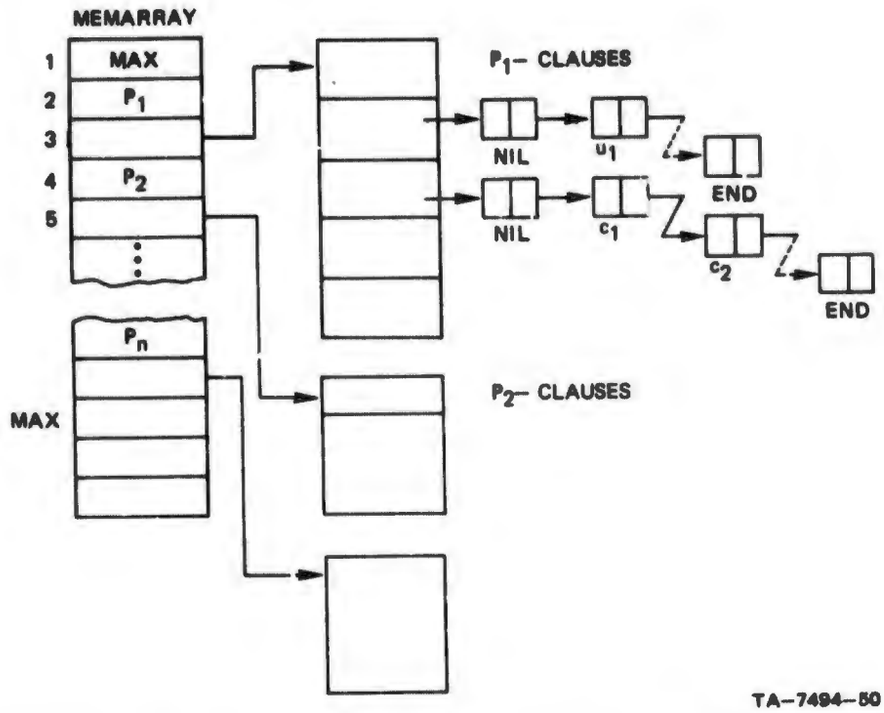
(3) Each list (NIL c_1 ... c_k END) begins with a NIL and ends with the atom END. The remaining elements are clauses of the same length as each other.

Note: CLAUSEARRAY is a global variable bound by the function START () and should not in general be tampered with. START () should not be called unless a fresh symbolic version of QA3.5 has been loaded.

b. MEMARRAY

MEMARRAY is an array containing all the clauses entered as axioms by the user. Every predicate letter currently in use appears on this array: The position on the array is the internal numerical representation given the predicate letter. Following the predicate letter is a pointer to an array (whose structure is the same as that of CLAUSEARRAY) of all the clauses in memory containing that predicate letter.

Axioms appear in MEMARRAY with the T-support marker set to T, as a bookkeeping device. When the axiom is fetched and used during a proof, this T-support is set to NIL as required, and the axiom together



with a pointer to its position in the subarray of MEMARRAY are stored on the list MCLAUSES so that the T-support marker can be restored after the proof.

MEMARRAY can be set to NIL either by executing the command RESET under the function QAS or else the top-level command (RSETMEMARRAY) - a function of no arguments.

4. Input

Inputs to QA3.5 are axioms and the theorem to be proved. These must be well-formed formulas (WFF's) in the first-order predicate logic. WFF's are entered in prefix form with the following QA3.5 operators allowed:

Logical operator	QA3.5 operator
~,¬	NOT
∧	AND
∨	OR
⇒,⊃	IF,IMP
⇔,≡	IFF,EQV
∀	FA
∃	EX

NOT is followed by one argument: the expression that is to be negated; AND (OR) is followed by any number of arguments, which are the expressions that are conjoined (disjoined).

IF and IMP are followed by two arguments: the antecedent and the consequent. Likewise, IFF and EQV must get two arguments: the two expressions that are equivalent.

FA and EX must also be followed by two arguments, the first of which is a list of quantified variables, the second is the statement over which the quantification takes place. Any input statement that violates these rules is not a WFF and will not be accepted by the system.

As an example, the logical expression

$$(\forall x,y\ (F(x) \wedge P(y) \supset (\exists z\ (\sim Q(x,y) \vee P\ (z)))))$$

would be typed to QAS as

```
(FA (X Y) (IF (AND (F X)(P Y))(EX (Z)(OR (NOT(Q X Y))(P Z)))))
```

When a WFF is entered, the system does a certain amount of translation (prenexing) and generates the HDR.

5. QAS Commands

Typing QAS(FILE) to LISP will cause the QAS executive to take control. If FILE is NIL (or missing), QAS will expect commands to be entered from the teletype; otherwise, QAS will get the appropriate file and read commands from it.

QAS allows the user to specify many different commands from several basic types.

The commands (with their arguments) under their general groupings are listed below.

a. Axiom Entering

S WFF--If WFF is well-formed, QAS enters it into MEMARRAY as an axiom; otherwise, an error message is generated.

SS WFF ROLE--Operates like S, except that it assigns a ROLE to the property list of the axiom; if ROLE is NIL, then the axiom is parsed according to the following format:

INPUT FORM	ROLE
$p[x_1;\ldots;x_n]$	FACT
$p[x_1;\ldots;x_n;SI]$	INITIAL-STATE
$\forall[x_1;\ldots;x_n]\ wff_1 \Rightarrow p[x_j;\ldots;x_n]$	(SUFFCOND P)
$\forall[x_1;\ldots;x_n]\ wff_1 \Rightarrow \exists y\ p[x_j;\ldots;y;\ldots;x_n]$	(EXISTENCE P)
$\forall[x_1;\ldots;x_n]\ p[x_1;\ldots;x_j] \Rightarrow wff_1$	(CONSEQUENT P)
$\forall[x_1;\ldots;x_n]\ p[x_1;\ldots;x_j] \Rightarrow q[x_{j+1};\ldots;x_n]$	(CONSEQUENT Q P)
$\forall[x_1;\ldots;x_n]\ p[x_1;\ldots;x_n] \Rightarrow q[x_1\ldots x_n]$	(SUBSET P Q)
$wff_1 \Leftrightarrow p[x_1;\ldots;x_j]$ is split into	
(a) $\forall[x_1;\ldots;x_n]\ p[x_1;\ldots;x_j] \Rightarrow wff_1$	(DEFINITION P)
and	
(2) $\forall[x_1;\ldots;x_n]\ wff_1 \Rightarrow p[x_1;\ldots;x_j]$	(NECCOND P)
any other WFF	AXIOM

The ROLE is printed out by LISTSENT (LIST within QAS) and UNWIND, as well as entered on the proof tree (see below).

AX AXNAME--Takes the name of an axiom previously put on AXIOMLIST (by AXIOMS), fetches the axiom, prints it on the teletype, and enters it into MEMARRAY; if the AXNAME is not on the list, then a message to that effect is typed out.

AXL AXNAMELIST--Similar to AX, but takes a list of axiom names and performs the steps for AX for each name. (These last two commands form one of the two types of axiom-naming available in the system.)

b. Theorem Entering

Q WFF--This command enters a WFF into CLAUSEARRAY and attempts to prove the WFF from axioms in MEMARRAY. If there are any existentially qualified variables in WFF, an attempt is made to generate an answer clause during a proof. This returns YES if the WFF is satisfied by the axioms, NO if the WFF is not satisfied by the axioms, NO PROOF FOUND if various bounds are exceeded in the course of the proof. If a proof is found and an answer clause is generated, the answer is printed.

TQ WFF--Exactly like Q WFF, except that it prints out the tracing of the proof as it goes; this does not print out generated clauses that are subsumed, but it can be made to do so by setting TR2 to T.

AQ WFF--Like Q WFF, except that it finds all answers to an existentially quantified WFF.

PROVE THNAME--Finds the WFF corresponding to THNAME on AXIOMLIST, prints it, and attempts to prove it in a fashion exactly similar to Q.

NPROVE THNAME--Exactly like PROVE, but negates the theorem before attempting the proof.

TPROVE THNAME--Like PROVE, but the proof is traced.

TNPROVE THNAME--Like NPROVE, with proof-tracing.

APROVE THNAME--Like PROVE, but attempts to find all answers to the question.

ANPROVE THNAME--Like APROVE, but the theorem is negated before the proof is attempted.

c. Clause Deletion

RESET--Clears MEMARRAY of all clauses.

FORGET P N--Deletes the Nth axiom stored under predicate P. If N is ALL, all axioms stored under P are deleted.

FORGETC P N--Deletes the Nth clause stored under predicate P.

d. Clause Listing

LIST P--Lists all axioms in MEMARRAY stored under the predicate P.

LISTC P--Lists all clauses in MEMARRAY stored under the predicate P.

LISTN NAME--Prints the clause named NAME (this is used with axiom naming).

e. Input and Output to Permanent Storage

AXIOMS P

AXN1 AX1

AXN2 AX2

: :

AXNM AXM--Builds up AXIOMLIST (which is an association-list). If P is T, AXIOMLIST is first set to NIL and then pairs of the form (AXNI · AXI) are inserted into it. If P is NIL, the pairs are placed at the front of the old AXIOMLIST. AXIOMLIST is a global variable.

WRITE file--Writes the commands or S-expressions following file and up to but not including STOP into file. If file is a legitimate external file name, then that file is used; otherwise, the file is on the property list of atom file.

CREATE file--Puts statements in MEMARRAY into file in the format (S AX1 S AX2 ...); file is added to the QAS file directory.

RUN file--Reads commands from file and executes them.

QAS--Sets file to NIL and begins reading from the teletype.

EDIT file--Allows editing of file using the LISP editor.

FILES--Lists the current file directory.

TREE treename--Saves the current proof tree on the atom treename.

f. Status

STATUS--Prints out the setting of various indicators in the system.

Q4STATUS--Prints out some of the more esoteric system variables.

g. Proof Output

UNWIND--Prints out the proof, along with a set of statistics indicating the amount of work that was done in the course of the proof.

h. Strategies

STRATEGY strat model--Allows the user to choose a search strategy. If AF is specified, the ancestry-filter strategy will be used (and no model should be typed in). If strat is MODEL, then model must be specified; model can be specified as ANL (all negative literals), APL (all positive literals), or a list of positive and negative literals. Strat can be a list of AF and MODEL in order to use a combination of the strategies. TSUPP is always used despite what other strategies are specified. If strat is NIL, then the strategy used will be TSUPP and unit-preference.

i. Miscellaneous

COMMENT--Allows the user to type in comments in the proof.

EXIT--Returns control to EVALQUOTE, and leaves QAS.

CONTINUE--Used when the proof terminates because MAXLEV was reached and sets MINLEV to the current value of MAXLEV, and MAXLEV to MAXLEV + 2; then continues the proof.

6. Other Capabilities

QA3.5 has several facilities that are handled by using the E command to QAS rather than by direct commands; some of the features are listed below and discussed in subsequent subsections.

(1) Property Lists may be associated with any clause. A set of functions for manipulating these lists has been written.

(2) Axiom and Clause Naming--A name can be associated with each axiom, and a mnemonically similar name associated with each of its clauses.

(3) Predicate and Function Evaluation--The user can associate with logical predicates and function a LISP function that will "evaluate" a literal and return T, NIL, a literal, or a value. The evaluation often dramatically improves the search time for a proof.

(4) Syntactic Symmetries--Two forms have been added to the unification and subsumption tests to enable resolution between literals that have permuted arguments or a list of identical (unordered) arguments.

(5) Human-Directed Proof Guidance--Several functions have been written to enable a user to "step" through a proof by attempting specified resolutions, as well as "killing" undesired clauses.

(6) Statistics--Certain interesting memory statistics can be printed out.

(7) MEMARRAY I/O--New functions have been written to simplify memory listings and to dump out and read in the complete MEMARRAY.

(8) Clause Size--The maximum length of clauses handled by the system is now a variable.

(9) Memory Map Functions--Simple functions have been written to apply an arbitrary LISP form to each clause in MEMARRAY, a subarray, or CLAUSEARRAY, and return a value.

(10) Factoring--A factor-checking function and answer-factoring feature have been written.

(11) Stopping a Proof--A proof can be stopped and resumed later.

a. Property Lists

A clause (MEMARRAY axiom or QA3 resolvent) can have a property list. Six standard items are stored by position or computed directly and referenced by the following flags: literals of a clause (LITS), T-support (TSUPP), level (LEVEL), history (HIST), length of a clause (LENGTH), and answer clause (ANSWERC). Other items are stored with an associated flag on the property list.

CLAUSE-FORM ← ((TSUPP LEVEL HIST ANSWERC PROPERTY-LIST) $lit_1 \ldots lit_k$)

(TSUPP LEVEL HIST ANSWERC PROPERTY-LIST) = HEADER = CAR(C)

$lit_1 \ldots lit_k$ = LITERALS = CDR(C)

PROPERTY-LIST ← $((prop_1 \ . \ val_1)(prop_2 \ . \ val_2)\ldots)$

Use:

(1) ptc[c;prop;val] places the value val under the flag prop on the property list (in the header) of c.
gtc[c;prop] retrieves the value of prop.
ftc[c;prop] deletes the pair (prop . val). Only flagged items can be deleted.
The preceding functions call pt[c;prop;val], gt[c;prop;val], and ft[c;prop;val] to deal with flagged properties.

(2) A clause c in the preceding function is represented explicitly. If clause naming is in effect (see below), one can call eval[ptc[clausename;prop;val]].
If a user breaks into a proof and wishes to mark a clause numbered N in the search, he can call (PTC(CAR(LASSOC N TRLIST))prop)val))).

(3) If one does not have clause naming, the following functions perform the same effect:

putc[predletter;n;prop;val]

getc[predletter;n;prop]

fgtc[predletter;n;prop]

where n is the number of the desired clause as listed (LISTC or listclauses[predletter]) under the predicate letter. Each of these uses the function nthclause[predletter;n] which returns the nth clause listed.

(4) Many of the features described below use property lists--including axiom naming, parsing, and proof guidance.

b. Axiom and Clause Naming

A user may specify a name for an axiom if it is of the form AXN, or he may have the system generate an axiom name of similar form for an axiom input via ENTER. Also, each clause in the prenex version of the axiom can be assigned a name of the form AXN-J. Thus AX10, an axiom yielding three clauses, can generate AX10-1, AX10-2, and AX10-3. Each clause name is stored under the clause property list (see below) under the flag NAME. Each WFF or clause is the value of the appropriate atom. In case of the clause, the atom value, e.g., CAR(AX10-2), accesses (HDR lit_1 lit_2 ... lit_j)--a list of axiom names is stored under the atom AXNAMELIST.

Use:

(1) For axiom naming, AXNAMING ← T. For clause naming, CLNAMING ← T.

(2) ENTER a WFF from the QAS or LISP executive as in the past. All the work is done within ENTER,

and a list of clause names followed by the axiom names is printed out as a side effect (but not returned as a value).

(3) The QAS FORGET functions work as before, but do not delete axioms or clauses from the system, since they are pointed to by top-level atoms. To delete an axiom or clause from the system (but not from MEMARRAY) call:

(a) delax[axname]

(b) delaxl[(AX1 AX2 ... AXN)] or delaxl[j;k] to delete $AX_j, AX_{j+1}, \ldots, AX_k$. (Note that delaxl[10;13] and delaxl[(AX10 AX11 AX12 AX13)] have the same effect.) These functions reset all the clause names (e.g., AX10-3) and axiom names to NOBIND and delete the axiom names from AXNAMELIST.

(4) AXCTR is a top-level atom, which serves as a counter for axiom names.

(5) If an axiom is deleted from MEMARRAY and re-entered, a new set of names and new clauses are generated. A function ENTERAX[axname] that will simply reinsert the old clauses in MEMARRAY is contemplated but unimplemented.

Note: Since a WFF and its descendent clauses are bound to atoms, their structure can be easily modified with the atom editor EDITV.

c. Predicate Evaluation

A user can associate a LISP function with any predicate to evaluate literals during search. MAKECLAUSE can be flagged to call evc[clause], which checks each literal of a newly constructed resolvent and allows options to evaluate the literal if its sign is positive, negative, or both, or if it is a ground literal, or not yet fully instantiated. If function predfn is associated with predicate pred,

predfn[lit] is evaluated as follows:

predfn[lit] =	T	delete clause
	NIL	delete literal
	lit	no change
	lit_1	substitute lit_1 for lit within the clause.

If pred has been specified for both positive and negated literals and predfn[lit] returns T or NIL, predfn[¬ lit] will return ¬ predfn[lit]. Furthermore, even if pred is specified only for evaluation on negated literals, predfn[lit] should be written for positive literals and the system will negate the result.

Use:

(1) Set EVCHECK ← T. For each function predfn associated with a predicate pred, execute evalpred[pred;sign;predfn;groundp].

If sign = POS only positive literals are evaluated
= NEG only negated literals are evaluated
= BOTH literals of both signs are evaluated.

If groundp = T the literal is evaluated only if it is in ground state
= NIL evaluated regardless of instantiation.

(2) Evalpred places the flag (EVALPOSP(predfn.groundp)) and/or the flag (EVALNEGP(predfn.groundp)) on the property list of the atom pred. Executing evalpred[pred;sign;NIL] will cause evaluation to cease for the case specified by sign.

Example: evalpred[GREATER;BOTH;GREATERP1;T] will cause the LISP function $greaterp_1$[lit] to be associated with the predicate "greater." If (GREATER J K) is fully instantiated (since groundp=T), $greaterp_1$[(± P J K)] would be evaluated. Note that the LISP function greaterp[x;y] cannot be

used, since it requires two arguments. One could define greaterp$_1$[lit] := greater[cadr[lit];caddr[lit]]. Executing evalpred[GREATER;POS;NIL] causes only literals of the form (-P J K) to be evaluated by greaterp$_1$[lit]. evalpred[GREATER;BOTH;NIL] ceases all evaluation by greaterp$_1$[lit].

d. Syntactic Symmetries and Generalized (Set) Unification

Occasionally a user encounters predicates that obey syntactic symmetries. In the past, the only way to represent these symmetries was to write a special axiom, e.g., $\forall_x \forall_y$ line[x;y] $\Rightarrow$ line[y;x]. The functional evaluation described in the preceding section has been used to implement a symmetric match function (PERMATCHF) associated with the function p[x_1;...;x_n], for arbitrary n. Any two equal, but unordered, sets will unify if they are represented as the arguments of the function ℓ[x_1;...;x_n].

Examples:

$\forall$abc point[a] $\wedge$ point[b] $\wedge$ point[c] $\wedge$ k-distinct[ℓ[a;b;c]]

$\Rightarrow$ plane[ℓ[a;b;c]]

$\forall$abc line[p[a;b]] $\wedge$ point[c] $\wedge$ not[on[c;p(a b)]] $\Leftrightarrow$ triangle[p[a;b;c]]

The preceding axioms exemplify a single use of these devices. Now consider the following clauses:

C1. line[p[A;B]]

C2. line[p[B;A]] triangle[p[B;A;x]]

C3. $\neg$ triangle[p[C;B;A]]

A and B are constants; x is a variable.

Note: line[A;B] and line[B;A] will not unify, but

R1 = C1 $\times$ C2 = triangle[p[B;A;x]]

R2 = R1 $\times$ C3 = NIL with θ = A/A; B/B; C/x .

Any two terms $r[a_1;\ldots;a_m]$ and $p[b_1;\ldots;b_n]$ unify only if n = m and there exists a cyclic permutation of b's that unifies with the a's.

Use:

(1) For syntactic symmetries, set P ← T. For unordered sets set L ← T.

(2) P is associated with permatchf[ ]. L is associated with tsubsume[ ]. These function names are stored on the property lists of P and L under the flag EVALFN.

(3) Write axioms with the appropriate function segments.

e. Human-Directed Proof Guidance

Occasionally a user will want to step through a proof to see whether a given set of axioms is adequate or see the form of a possible proof. The user can now specify two clauses with two of their literals and attempt a resolution. During an automatic research a user can stop the system, denote certain resolvents or axioms as KILLED, and when the system is spurting automatically it will behave as if those clauses no longer existed.

Use:

(1) To stop a search in progress, hit a H^c. To prevent any search, but also enable QUESTION to put the prenex version of the question on CLAUSELIST, execute: BREAKIN(FN1(BEFORE UNITRESTEST)T). Set TR1←T at least. (TR2←T is optional.)

(2) TRYR(C1 L1 C2 L2) attempts to resolve C1 and C2 on L1 and L2, which must be unifiable. If L1 and L2 do not unify, an error message is returned. All the arguments of TRYR are atomic. TRYR(AX11-1 3 53 2) attempts to resolve the clause named AX11-1 on its third literal with

the clause numbered 53 in the proof search, on its second literal. Some extra unit resolvents might be added automatically by UNITSECTION.

(3) KILL(N) sets a flag (KILL T) on the property list of the clause numbered n in the search. Any clause with a KILL flag will never be resolved.

UNKILL(N) reverses the effect of KILL(N); The clause "reappears."

(4) A use who, at the outset of a proof, wants to add certain clauses from memory to CLAUSEARRAY, should execute either an H^c during search or BREAKIN(QUESTION(BEFORE COND 4)T) prior to search. Within the BREAK execute: (PREF pred n len s) to add the nth clause of length len filed under predicate pred tc CLAUSEARRAY and if s = T, the clause will get T-support.

pref[ON;3;2;T] will place the third 2-clause filed under predicate ON on CLAUSEARRAY with T-support. prefl[pred($(n_1 len_1 s)(n_2 len_2 s_2)\ldots$)] in an n-lambda which calls pref[n_1;len_1;s], pref[n_2;len_2;s], etc.

Caution: A user guiding a proof who does the complete proof by hand and reaches a contradiction, should execute (UNWIND PROOF) or (TREESAVE TREENAME) immediately. Returning to FN1 to allow the system to stop on its own accord will cause a deep error that will send the user back to the LISP executive and, as a result, erase CLAUSEARRAY.

f. Statistics

Occasionally a user wants a more refined description of a data base than the gross number of axioms. Executing MEMSTAT[ ] enables the

user to obtain the following printout for each predicate in MEMARRAY:

PREDICATE P1

THERE ARE X_1 clauses of length 1

THERE ARE X_2 clauses of length 2

:

THERE ARE X_n clauses of length n

IT REFERENCES THE PREDICATES (P3 P4 P17 P21) .

The following printout is global:

MEMORY STATISTICS

DATE

:

There are N clauses in <u>all</u> .

Use:

Execute MEMSTART

g. <u>MEMARRAY I/O</u>

For a large data base, the use of LISTSENT(PRED) to obtain a complete listing of WFF's is all too time-consuming. Moreover, since WFF's are cross-referenced under each predicate in the WFF, each WFF may appear four or five times. BRIEFLIST[(p_1 p_2 p_3 ... p_n,] prints out a "minimal list" of all the WFF's stored under the predicate names p_1, p_2, ..., p_n. The role of each WFF is printed out along with a number.

When axioms yielding several clauses each appear and the number of WFF's begins to exceed 25, using the standard ENTER (or run[file]) can be quite time-consuming.

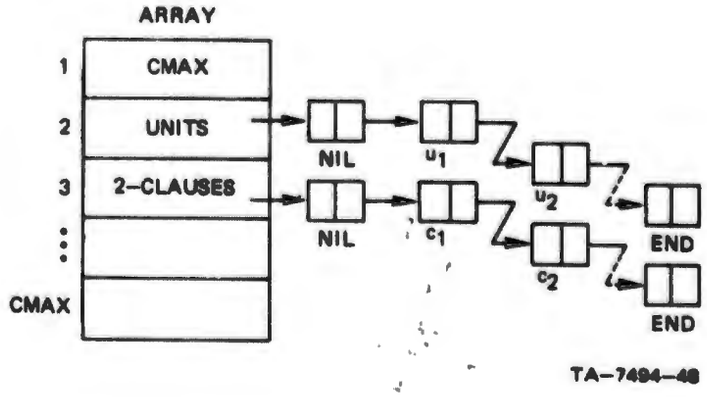
DUMPOUT[filename] copies the complete MEMARRAY onto the file specified along with the settings of SKOLEMS and AXCTR to reset the skolem function generator and the axiom-naming generator. LOAD[filename] calls FILLIN, which clears the current MEMARRAY and fills it with the dumpout file.

h. Clause Size

The current system handles clauses of length ≤ 9. If one wishes to handle clauses of substantially shorter length--e.g., ≤5--and wants to save space, or if one wants to occasionally handle larger clauses and know when one is entered, then the CLAUSESIZE facility is appropriate. No user changes are necessary, although several QA3.5 functions have been modified. A top-level parameter CLAUSESIZE is set to the desired clause size. Subarrays of MEMARRAY and CLAUSEARRAY are generated to handle clauses of appropriate length. If a WFF is entered that generates a clause of greater length than CLAUSESIZE, the entry is aborted and a message with length information is printed out to the user.

i. Memory Map Functions

Two simple functions have been written to apply an arbitrary function to each clause--e.g., ((HDR lit_1 ... lit_j)) in MEMARRAY, CLAUSEARRAY, or any axiom-styled subarray. Mapmemc[fun] applies fun to each clause in MEMARRAY. maparc[array;fun] applies fun to each clause stored in array, which is assumed to be of the form:



Use:

(1) delprop[prop]:= mapmemc[function λ[c] ftc[c;prop]]
deletes prop from each clause in MEMARRAY.

(2) names[array;x].= prog[[x];maparc[array;function[λ[c;y];
setq[y;gtc[c;NAME]];
cond[membly[x] → NIL;T → setq[x;cons[y;x]]]return[x]]]].
Names[array] returns a nonredundant list of the
names of all the clauses is stored in array.

j. Factoring

The value of an existential variable is stored in clausal form under the flag ANSWERC on a resolvent header. When this clause has length >1, factoring may give the most specific answer from a possible set.

Use:

Set FACTORANSWER ← T.

Given two clauses C1, C2, and ℓ1 and ℓ2 of C1 and C2, respectively, a function checkres[C1;C2] determines whether the resolvent of C1 and C2 should be computed. Likewise, a function factorcheck[C] may be written by a user to check for necessary conditioning for factoring before factoring is attempted. The function name is properly embedded within FACTORSECTION but each user must specify his own function definition.

Use:

Set FACTORCHECK to T.

k. Stopping a Proof

It is possible for a user to stop a proof and then continue later from the same place. To do this, the user interrupts with H^C. Then set STOP to T and type OK. The proof will continue until it reaches a convenient place to stop.

To restart, execute QUESTION with no arguments.

7. Variables

QAS references several global variables. Most of these control options; some are set by functions in the system, while others must be set explicitly by the user, via the SET function.

These variables are:

Variable	Value	Result
ABMAXLEV	n	Specifies the absolute maximum level bound
ANCESTORTEST	T	Ancestry-filter strategy is in effect
	NIL	No ancestry-filter
ANSSUBSM	T	Prevents a clause from being entered on CLAUSEARRAY when its answer clause is subsumed by another
	NIL	No check
ANSWERTEST	T	Causes a trace of all universally quantified variables in the negation of the theorem, after the proof is finished
	NIL	No action
ANSWERTEST1	T	Causes the answer clause to be printed during a trace
	NIL	No action
ANSWERTEST2	T	Causes the answer clause to be printed during an unwind
	NIL	No action
AXNAMING	T	Axiom naming is in use
	NIL	No effect
CLNAMING	T	Clause naming in effect
	NIL	No effect
ENTERUPT	T	Allows user to intercede in clause entering
	NIL	No effect
EVCHECK	T	Initiates predicate evaluation
	NIL	No effect
FACTORANSWER	T	Causes an attempt to factor the answer clause
	NIL	No effect

Variable	Value	Result
FACTORCHECK	T	Allows user to define a predicate to control factoring (in FACTORSECTION)
	NIL	No effect
GREENGROW	T	Allows user to continue a proof
	NIL	No attempt made to continue
KEEPCLAUSESWITCH	T	Allows user to decide which clauses to keep
	NIL	No effect
KEEPANSUNITSONLY	T	Retains only unit answer clauses
	NIL	No effect
MAXDEPTH	n	Controls depth of function nesting
MAXLEV	n	Controls level of proof tree
MODELTEST	T	Resolution is with respect to a MODEL
	NIL	No effect
NOONLY	T	Causes QAS to attempt only NO answers
	NIL	Attempts both YES and NO answers
PROOFTIME	T	Causes a time message to be printed at end of UNWIND
	NIL	No time message
SKIP	T	Used to continue a proof
	NIL	No continuation
TRSW	T	No effect
	NIL	TRLIST is built up during proof, but no printing (tracing) is done
TR1	T	Builds TRLIST (list used in tracing) and traces proof
	NIL	TRLIST is not built and no tracing is done
TR2	T	Causes clauses subsumed by other clauses to be printed during tracing when TR1 = T
	NIL	Clauses subsumed are not printed
YESONLY	T	QAS attempts only YES answers
	NIL	Attempt both YES and NO answers

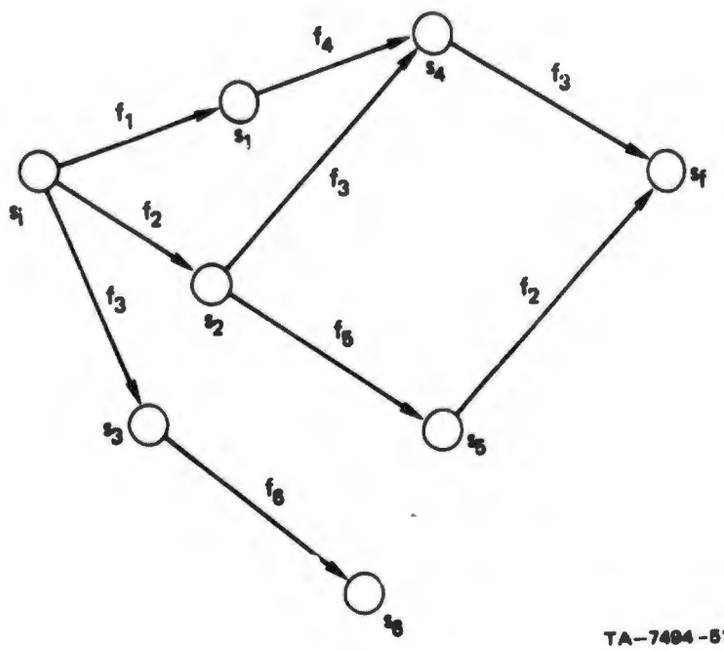
Appendix B

A BRIEF DESCRIPTION OF THE SITUATION CALCULUS

Appendix B

A BRIEF DESCRIPTION OF THE SITUATION CALCULUS

The Situation Calculus (SC) is a formal method for describing a class of transformations on a set of states in the first-order predicate calculus. Its advantage lies in its ability to represent transformations in first-order logic (where extensive computer programs are available for logical deduction), where higher-order or modal logic would normally be used. It is often convenient to think of the state space and its accompanying transformations as a directed graph. For example,



TA-7494-51

where s_i and s_f are the initial and final states respectively, and the f_i $i = 1, 2, \ldots, 6$ are transformations that map states onto states. In the directed graph any problem of going from the initial to the final state can be formulated as follows: Does there exist a path from s_i to s_f? In the above graph $f_1f_4f_3$, $f_2f_3f_3$, or $f_2f_5f_2$ might be given as a possible solution.

More formally, the situation calculus is a five tuple:

$$SC = \{\mathcal{A}, \mathcal{P}, \mathcal{F}, S, \Omega\} \quad ,$$

where

$\mathcal{A}$ is a set of axioms;

$\mathcal{P}$ is a set of predicates describing states;

$\mathcal{F}$ is a set of primitive functions that map states on to states, $\mathcal{F}: S \rightarrow S$;

S is a set of states; and

Ω is a universe of discourse.

All n-ary predicates are represented in the SC as n + 1-ary predicates with the n + 1st argument being the universal state variable, s. For example, P(x) becomes P(x,s) in the SC.

Axioms in the SC take the form

$$P(x,s_1) \wedge F(x,s_1) = s_2 \Rightarrow Q(x,s_2) \quad P,Q \in \mathcal{P} \quad ; \quad F \in \mathcal{F} \quad ; \quad s_1, s_2 \in S \quad , \qquad (1)$$

where P is a predicate (or possibly a conjunction of predicates) over a set of arguments (represented by the vector x) describing the relevant features of the situation s_1; Q is a similar predicate describing the situation s_2, which obtains when F is applied to s_1; and thus, by convention, $F(x,s_1) = s_2$. Simplifying the axiom schema (1) by explicitly incorporating the fact that $F(x,s_1) = x_2$ and universally quantifying x and s, we may write

$$(\forall x,s)\{P(x,s) \Rightarrow Q(x,F(x,s))\} \quad . \qquad (2)$$

Expression (2) is then the general form in which SC axioms are written, and we may think of P in our problem solving process as describing the boundary conditions that must be satisfied in any state for the function F to be applicable. Alternatively, and perhaps even more simply, one

may view the axioms as a set of productions or rewrite rules of the form $P \rightarrow Q$. In addition, we must have some initial axiom that describes the initial conditions or the initial state, s_i of the form

$$P(c, s_i) \; c \in \Omega \; ; \; s_i \in S \quad . \tag{3}$$

where c is a vector of constants and P (possibly a conjunction of predicates) is true of the initial state.

A problem in the SC then takes the form

$$(\exists s_f) \; \{R(c, s_f)\} \quad , \tag{4}$$

or "Does there exist a final state such that R(c) is true in that state?" A solution takes the form

$$s_f = F_n(c, F_{n-1}(c, \ldots, F_3(c, F_1(c, s_i)) \ldots)) \quad , \tag{5}$$

or more simply,

$$s_f = F_n F_{n-1} \cdots F_3 F_2 F_1(c, s_i) \; F_i \in \Im \; i = 1, 2, \ldots, n < \infty \quad ; \tag{6}$$

i.e., a composition of functions applied to the initial state that will yield the final desired state such that $R(c, s_f)$ is true in that state.

A more complete presentation of the method for using first-order predicate calculus theorem proving in solving problems can be found in Ref. 5.

Appendix C

THE N-TUPLE STORAGE SYSTEM

BLANK PAGE

Appendix C

THE N-TUPLE STORAGE SYSTEM*

The N-tuple Storage System (NSS) is a means of storing and retrieving ordered n-tuples of information. It exists as a collection of functions within a LISP operating system.

This appendix is in four sections. The first describes the functional (i.e., hopefully implementation-independent) specifications of the NSS. The second section briefly outlines the implementation that is currently planned for the SRI Artificial Intelligence Group's PDP-10. The third section records some of the design considerations that have led to this implementation. The fourth section describes additional functions added to the NSS.

1. Functional Specifications

a. Introduction

An ordered n-tuple is an ordered collection of n objects,

$$\left(x_1, \ldots, x_n\right) \quad .$$

Throughout this appendix, we portray ordered n-tuples in the form:

$$\boxed{X = \left(R\ A_1 \ldots A_m\right) \ , \ m = n - 1} \quad ,$$

omitting the delimiting commas in the manner of LISP and giving preferred status to the first element of the n-tuple. (This notation is suggested by the idea that R is frequently a relation, and the A's are the arguments of the relation. However, the idea is not a canon, and (JOHN BROTHER DOUG) is a perfectly good n-tuple.)

---

*By John H. Munson.

The top-level functions of the NSS are:

(NTSTORE X), which "stores" X;
(NTQUERY X), which finds stored n-tuples matching X;
(NTERASE X), which erases X from storage.

For each of these functions, X must evaluate to a LISP S-expression (the n-tuple) satisfying the following:

(1) X is a nonatomic list structure;

(2) (CDR X), which is the list of A's, must be nonnull,

(3) (CAR X), which is R, must be a nonnumeric atom (in other words, a LISP identifier).

If any of these conditions are not met, the message "(NTUPLE FORM ERROR)" followed by X is printed, and the top-level function takes no action and returns NIL.

The above conditions ensure that X evaluates to a legitimate n-tuple, with $n \geq 2$. It has an R that is an identifier, and at least one A. The A's may be any LISP expressions that may appear as elements of a list. (The reasons for restricting R are covered below.)

In what follows, we shall refer to "whether two A's are identical." This is meant in terms of the LISP predicate EQ. EQ is true for occurrences of the same identifier, equal numbers of the same type, and expressions that evaluate to the same pointer. Caution: EQ is not true for equal numbers of different types, nor for separately generated occurrences of identical-looking list structures. Thus,

(EQ(QUOTE(X Y))(QUOTE(X Y))) = NIL .

b. (NTFILE X)

This is the filing or "store" command. It checks whether an n-tuple identical to X (i.e., of exactly the same length and with R and each A identical to the corresponding member of X) is already stored. If so, NTFILE does nothing and returns NIL. If not, NTFILE stores X and returns T.

c. (NTERASE X)

This function checks whether an n-tuple identical to X is already stored. If so, NTERASE erases the n-tuple from the storage structure and returns T. If not, NTERASE does nothing and returns NIL.

The "erasure of an n-tuple from the storage structure" does not affect the n-tuple itself, nor any pointers to it that are external to the storage structure maintained by these routines. Only pointers within the (invisible) structure are deleted. However, the structure's pointers are to the originally filed n-tuple, not to a copy of it; hence, any tampering with the n-tuple itself (through RPLACA, RPLACD, NCONC, etc.) will invalidate the storage mechanism and cause errors.

NTFILE and NTERASE use a common identity check. Since the identity of two n-tuples is a well-defined concept, it follows that an n-tuple can only be stored once in the system at any given point in time, and that NTERASE will erase the only occurrence of the n-tuple, if there is one.

d. (NTQUERY X)

This function searches the storage system for all n-tuples that match X. If none are found, NTQUERY returns NIL. If any are found, NTQUERY returns a list of them. If X is fully specified--that is, if none of the A's of X is the LISP identifier ALL--then the only n-tuple that can be returned is the (unique) n-tuple identical to X. Any or all of the A's of X, however, may be the identifier ALL, in which case each stored n-tuple that is identical to X in R and the remaining A's is returned. In the extreme case,

X = (R ALL ... ALL)

returns a list of all n-tuples of the proper length stored under R. N-tuples of all lengths may be stored under the same R, and n-tuples of different lengths never match.

e. Comments and Fine Points

One could presently use ALL as an argument of an n-tuple in NTFILE and NTQUERY. Once stored, it behaves in the same manner as any other identifier. However, this practice should be avoided because in the future we might allow a stored ALL to match any corresponding A in X. This is not implemented at present because it would increase the search time and add a little program complexity, since we would have to search under R/ALL whenever the search under R/A failed.

Do not use ANY as an argument of an n-tuple. It may be used in the future to give matching properties similar to ALL, but with the added feature that if any ANY appears in X, the search will terminate after the first match is found. Since the same result can be obtained (albeit while going through the full search) by

(COND((SETQ Z(NTQUERY X))(LIST(CAR Z)))) ,

the ANY-feature has not been implemented until the need for it is established.

The NSS demands that R always be specified. This greatly facilitates an efficient implementation. It is expected to be a rare situation, at least in our planned work, in which an always-present R cannot be chosen. If such a situation does arise, it can be handled by a convention that stores an n-tuple as an (n + 1)-tuple with a fixed, dummy, R.

Since an A may be any LISP expression, the feature of allowing a component of an n-tuple to be, in turn, an n-tuple is immediately accommodated.

2. Implementation

a. Structure

The storage structure for all n-tuples with a given relation R is kept on the property list for that R, under the property NTUPLES. Thus, the search for an n-tuple is immediately fanned out to the proper R, accomplishing the first reduction in search space. The value stored on the property list is a nested list structure whose top-level positions correspond to the argument positions of n-tuples:

((list for first arg's) ... (list for Mth arg's)) ,

where M is the maximum of m, or (n - 1), over all the n-tuples that have been stored under this R. Note that R itself does not appear within the structure.

The list for an argument position may be NIL if nothing is stored there, but it is required to keep place. Otherwise, the ith position-list is a list of even length:

$\left(A_1^i \text{ (list of reduced n-tuple's)} \ \ldots \ A_\omega^i \text{ (list of reduced n-tuple's)}\right)$

Here, for each A, the list following it includes pointers for all currently stored n-tuples for which that A appears as the ith argument with R; however, these pointers are to reduced n-tuples with the R removed (in other words, to the CDR's of the original n-tuples). Thus, the command

(NTFILE(QUOTE(R A B)))

given to a fresh system will generate the structure

((NONREF NIL A((A B))) (B((A B))))

The dummy argument NONREF, with an initially NIL list of reduced n-tuples, is always established as the first argument in the first-position list, and remains there. Its use will be explained shortly.

b. Referable and Nonreferable Arguments

Arguments, which may be any LISP expressions, are divided into two classes according to the predicate REFERABLE. Currently, REFERABLE = T for all identifiers (nonnumeric atoms) other than ALL, and REFERABLE = NIL otherwise. Entries are made in the storage structure only under referable arguments, with this exception: If the first argument of an n-tuple is nonreferable, a special entry is made under NONREF. If a second, third, or other argument is nonreferable, no entry is made in that position. (This ensures that every n-tuple is stored somewhere in the first-position substructure, and can be found there even if a later target n-tuple X has ALL's in all the referable positions.)

Thus, to continue the example above, a second command

```
(NTFILE(QUOTE(R 99(P Q))))
```

would cause the structure to be changed to

```
((NONREF(99(P Q)) A((A B))) (B((A B))))
```

(Whether we have chosen the best definition of REFERABLE, and whether the distinction is even worthwhile, are questions that only experience will answer. The distinction is clearly not required, as we have ensured that any stored n-tuple can be found by an X with any number of ALL's. At one extreme, we could do away with REFERABLE and store every n-tuple under all its arguments; at the other extreme, we could simply store all the n-tuples under R, perhaps with some indexing scheme, and search the entire applicable structure, testing the identity of each n-tuple as we go. We have retained REFERABLE because it provides some control over what is stored and what must be expanded in search. Furthermore, if in the future we changed the storage mechanism to some

other scheme--such as hash coding--we might want to restore the original sense of REFERABLE, which was to determine under which arguments the n-tuple was stored and could be reaccessed, and under which it was not.)

c. Searching and Matching

With the above background, the procedure (function NTUPLOC) for finding all n-tuples that match X can be described simply. X is scanned until a referable argument is found, or X is exhausted. In the former case, the corresponding position substructure is searched for an occurence of the referable argument, and if it is found the associated list of reduced n-tuples is scanned, testing each one for a match with the reduced X. In the latter case, when X has no referable argument, the search for matches is carried out over the whole first-position substructure, including NONREF. In either case, when a match is found, the reduced n-tuple is appended to R and the result added to a list, originally NIL, that is returned as the value of NTUPLOC. Thus, if X has any referable argument, the search space is divided two times: first by R, then by the referable argument.

All three functions NTFILE, NTQUERY, and NTERASE begin by calling function NTUPCHK, which performs the form checks described in the first section. NTUPCHK also sets global variable NTUPREL to the relation R of X, NTUPPROBE to the argument list of X, and NTUPSTRUC to the storage structure under R. After execution of the functions, NTUPSTRUC is handy for viewing the resulting structure.

d. The Top-Level Functions

NTQUERY simply calls NTUPCHK and, if NTUPSTRUC is not NIL, calls NTUPLOC and returns the value returned by NTUPLOC.

NTFILE calls NTUPCHK and then NTUPLOC. If an exact match to X is found in storage, NTFILE returns NIL. Otherwise, it calls subsidiary routines NTFILE1 and NTFILE2, which go through the structure under R, making the appropriate additions and insertions.

NTERASE also calls NTUPCHK and NTUPLOC. If an exact match is found, as it usually should be, NTERASE and its subsidiary function

NTERASE1 go through the structure. Wherever a pointer to the matching (reduced) n-tuple is found, it is deleted and the structure closed up around it. However, if a list of n-tuples following an argument name is reduced to nothing, the portion

... argname NIL ...

is left in the structure; it is not excised. (This aspect could be changed, at the expense of some additional program complexity.)

Both NTFILE and NTERASE make use of the structure-changing LISP functions RPLACA, RPLACD, and NCONC to modify the storage structures in place, rather than make unnecessary copies.

e. Storage Requirements

All of the storage structures are maintained in regular LISP free storage space. The storage requirements are:

(1) (n - 1) PDP-10 words per n-tuple, for the original list of arguments (not a copy);

(2) From 1 to (n - 1) words for the pointers, depending on the number of referable arguments;

(3) Two words for each first appearance of a referable argument in a given position under a given R; and

(4) (n + 3) words for each R involved (using the maximum n for that R).

Thus, the requirements typically range from roughly n words per n-tuple (no referable arguments) to roughly 4n words per n-tuple (all arguments new and referable; common R).

We have no practical knowledge yet of the running speeds. The routines presumably have no monstrous inefficiencies, but they are doing list searches that will get long if the lists do. The routines could clearly be dominated in speed by a hash-coding scheme that has entries

for all combinations of specified arguments, but this seems expensive in terms of storage. This trade-off is discussed more fully in Part 3 of this appendix.

3. Design Considerations

The first question to be discussed is, Why did we choose this level of implementation effort? Why not more? Why not less? We can begin by agreeing that there is certainly a recurrent need to store ordered n-tuples of information. In our robot work, they might appear as world-facts,

(COLOR BOX1 BLUE)

(IN ROBOT ROOM3) ,

as theorems and axioms in the question-answering program,

(IMPLIES P Q)

as values of quantities,

(X WEDGE3 37.6)

(TILTANGLE -19) ,

and handles for accessing other forms of data,

(TVPICTURE 5 (a large array)) ,

and so on.

We want not only to store such information, but to access it with various elements specified at different times. Sometimes, it is "Where's the robot?" Other times, it is "Who's in ROOM3?" This means that the n-tuples cannot simply be stored using the value or property mechanisms

of LISP; there must be enough of a system added to facilitate cross-filing and argument-field matching. Of course, an associative memory is such a system, if only somebody would build an associative memory we could buy.

The NSS represents a minimal effort to achieve the desired storage and retrieval capability. In effect, it adds a new data type--the n-tuple--to LISP.

The NSS may be contrasted with far more ambitious projects that create entire systems incorporating these capabilities. Ash and Sibley[18] have focused on automating the extension of the retrieval process to include associations that depend on inferences supplied to the program. Feldman and Rovner[19] have created an entire language, LEAP, which is an extension of ALGOL to include association and several other features. The NSS, which is intended primarily as a storage mechanism for the robot programs, did not merit anything like the extensive effort involved in these other systems.

Furthermore, we did not wish to try to predict in advance the further features that users may want to add to NSS. In keeping the initial structure simple, and the programming effort modest, we are betting that additions can be easily made "on top of" NSS. Perhaps, the less fancy NSS starts out, the more easily it can be molded using the full power of LISP, which surrounds it.

The other big question is, "Why was NSS not built using hash coding?" First, it must be observed that the expected context of NSS has two features that diminish the attractiveness of hash coding. One is that we preferred to deal directly with n-tuples, whereas both TRAMP and LEAP begin with triples. Of course triples can be strung together to form n-tuples, but one may be concerned about the price to be paid in efficiency. To hash-code n-tuples directly, one must either store the hash addresses for all the approximately 2^n ways the specification of elements can be chosen, or else resort to some search after an initial entry.

The other feature is the expected occurrence of families of n-tuples with common elements. We will have, for example,

(IN ROOM3 ROBOT)

(IN ROOM3 BOX1)

(IN ROOM3 BOX5)

and so on. The system must return all of these to the query

(IN ROOM3 ALL)

A hashing scheme must either treat these as collisions, of which there will be undesirably many, or collect them in a sublist. The speed advantage of hashing begins to be lost.

In short, we did not make any extensive analysis of the possibilities of hash coding and its many variations, but in a brief study we did not find any clear advantage for the schemes we surveyed in light of the features noted above.

4. Additional Functions

The original top-level functions of the NSS are

(NTFILE X), which "stores" X;

(NTQUERY X), which finds stored n-tuples matching X; and

(NTERASE X), which erases X from storage;

where $X = (X_1 \ldots X_n)$ is an n-tuple.

In beginning to use the NSS, we quickly discovered that functions other than these three are generally better tailored to the user's needs. At some cost in efficiency, these functions have been implemented for the time being in terms of the basic functions listed above.

a. NTPURGE

(NTPURGE X) erases all n-tuples matching X, allowing substitutions where X has "ALL." It returns NIL.

The remaining functions are tailored to work with situations in which, for a given specification of all but one element of the n-tuple, the remaining element varies from time to time but there is only one such n-tuple stored at any one time. To produce an example from chess, we might have

(WHITE KNIGHT2 pos)

where pos specifies a rank and file, and since this knight has a unique position at any time, only one such n-tuple is ever in storage. For this type of situation, it is convenient to have tailored read and store commands.

b. NTREAD

(NTREAD X) returns the element--not the n-tuple--matching the first ALL in X (reading left to right) in the first n-tuple found in storage that matches X. As above, it is generally assumed that at most one matching n-tuple will be stored. If no match is found, NTREAD returns NIL.

c. NTSTORE

(NTSTOREi X) stores X after purging all n-tuples matching $(X_1 \ldots ALL \ldots X_n)$, where ALL is in the ith position.

In the chess example,

(NTSTORE3 (QUOTE (WHITE KNIGHT2 KB3)))

will purge any older positions and store the n-tuple, and

(NTREAD (QUOTE (WHITE KNIGHT2 ALL)))

will subsequently return the value KB3.

Appendix D

DECISION-TREE SOFTWARE

BLANK PAGE

Appendix D

DECISION-TREE SOFTWARE

1. Introduction

This appendix describes the decision-tree software used with our current scene-analysis programs. The heart of this software is an interpreter, which executes the low-level operators in a sequence determined by a symbolically described decision tree. The interpreter, however, is but one of four programs that allow a user to do the following:

(1) Define a language for use in describing a decision tree;

(2) Compile a program written in that language;

(3) Execute the compiled program on given input data; and

(4) Obtain a readable listing of the object code.

These are, respectively, the basic functions of the language symbol definition program, the translator, the interpreter, and the untranslator.

2. The Language Symbol Definition Program

The language used for defining the decision tree contains user-defined mnemonics. The translator requires a table of information about these mnemonics to recognize and interpret them correctly. The main function of the language symbol definition program is to provide such a table. The table is built from a list of mnemonics provided by the user.

The mnemonics are divided into two classes: those representing low-level operators to be executed and those representing directives to the interpreter. In the sequel, the former are called tests, and the latter are called direct actions. The mnemonics themselves must be an alphanumeric string starting with a letter; they can be of any length, but only the first three characters are used.

The output of the language symbol definition program is a set of formats that control the packing of compiled code and data throughout the system. Among other things, this allows the user to control the size of various data fields, and consequently the magnitudes of the numbers contained in them. Users of the program need to know the following information.

a. Input Data

The language symbol definition program requires two drum files as input:

(1) /SYMB/--This file contains three types of information:

(a) The number of direct actions included in the language being specified.

(b) A list of the mnemonics representing the direct actions used in the language.

(c) A list of mnemonics representing the tests used in the language.

The format of this file is:

⟨ number of direct actions (integer) ⟩ (CR)
⟨ direct action mnemonic #1 ⟩ (CR)
⟨ direct action mnemonic #2 ⟩ (CR)
⋮
⟨ direct action mnemonic #N ⟩ (CR)
⟨ test mnemonic #1 ⟩ (CR)
⟨ test mnemonic #2 ⟩ (CR)
⋮
⟨ test mnemonic #M ⟩ (CR)
END (CR).

Note that the last record on the file must be the word "END" followed by a carriage return. Omission of this will cause the program to enter a tight error loop.

The total number of mnemonics is currently limited to 32. This number may be changed by altering a dimension statement within the program.

(2) /PACKFMT/--This file contains formats that are used to control the packing of fields throughout the system. A format has five elements:

(a) Size of field (in bits) (1-24)

(b) Starting bit of source word (0-23)

(c) Starting word of source word list

(d) Starting bit of destination word (0-23)

(e) Starting word of destination word list.

Each element is represented by an integer and must be followed by a comma. Each format must be followed by a carriage return. Hence, the general format of /PACKFMT/ is:

⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, (CR)
⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, (CR)

⋮

⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, ⟨ int ⟩, (CR) .

In order to understand the meaning of the various fields in a format, a short description of the packing technique is in order. Fields are packed by a routine that performs a generalized move of a bit string from one location to another. The inputs to this routine are a format list (controlling the moves), a source string, and a destination string. Although the source and destination strings are considered to be bit strings, they are addressed by word and bit, relative to the start

of the string. If there were a true address that identified a field by its bit address relative to the start of the string, the word and bit addresses would be defined as follows:

```
WORD : = TRUE DIV 24
BIT  : = TRUE MOD 24 .
```

During the execution of the language symbol definition program, there will be a request for input from the controlling teletype by the message:

```
PRINT PACKING FORMATS?
```

The response to this request may be Y (yes) or N (no). If a positive response is indicated, the octal values of the formats generated from the information in /PACKFMT/ will be typed (one format per line).

b. Output

If a print of the packing formats is requested, then these will be listed on the teletype. This entails approximately two pages of listing. The packing formats may be requested by an appropriate response to the "PRINT PACKING FORMATS?" message (see section on input).

The mnemonics specified by the file /SYMB/ will be listed on the controlling teletype, along with a reference number. The mnemonics will be listed in two groups, one group consisting of the test mnemonics, and the other of the direct action mnemonics. The reference number listed with the mnemonic is the number that will represent that test or action in the translator, interpreter, and untranslator.

A drum file labelled /ITAB/ will be created. This file contains all of the information required by the translator for translating programs. Briefly, this information is:

(1) Symbol table of mnemonics (with reference numbers)

(2) Packing formats

(3) Number of mnemonics

(4) Number of direct actions.

c. Implementation Information

The language symbol definition program has two functions, and therefore consists of two logically separate phases. The first phase builds the packing formats, and the second builds the symbol table. During the first phase, the file /PACKFMT/ is read by a routine INFMT. This routine calls MOVFMT, passing the information contained in one record of /PACKFMT/ as a parameter. MOVFMT packs this into one word, and returns this word as its value. This process is repeated until all of the formats on /PACKFMT/ have been processed. During the second phase, the first element of /SYMB/ is read, and is stored as the number of direct actions (N). The remainder of the records on /SYMB/ are read. Each record must start with an alphabetic character, and it may contain only letters and numbers. Each record is truncated to three characters, and these are taken to be the mnemonic. Duplicate mnemonics are ignored, and a message to this effect is typed.

When these two phases are completed, the file /ITAB/ is created, containing first the packing formats, and then the symbol table information.

3. The Translator

The translator reads a symbolic description of a decision tree and translates it into a sequence of instructions that are meaningful and can be executed by the interpreter. The translator recognizes two types of statements: test statements and directive statements. The test statements are translated under a fixed syntax, whereas the user must specify the syntax to be used in translating directive statements. The user must provide the code in the translator for recognizing this syntax. A collection of utility routines are provided to minimize this task.

With the aid of these routines, one FORTRAN statement is frequently sufficient to recognize a construct.

A user of the translator must know the following information.

a. Input Data

The translator requires two drum files as input:

(1) /ITAB/--This file is produced by the execution of the language symbol definition program.

(2) /PROG/--This file contains the source code to be translated. There are two general types of statements that may be used as source statements:

(a) Test statements are used to describe the tests to be executed in directing the flow of the program. These have the following general format:

⟨ test ⟩ ⟨ true branch ⟩ ⟨ true action ⟩

⟨ false branch ⟩ ⟨ false action ⟩ .

The metalinguistic symbols have the following meanings:

⟨ test ⟩. This is a test mnemonic as specified in the language symbol definition program.

⟨ true branch ⟩. This must be a ⟨ label ⟩.

⟨ true action ⟩. This may be either in integer or a mnemonic representing an indirect action (the mnemonics used for direct actions may also be used to specify indirect actions).

⟨ false branch ⟩. Same as ⟨ true branch ⟩

⟨ false action ⟩. Same as ⟨ true action ⟩.

(b) Direct statements describe direct actions to be taken in the course of executing the program.

The general format is:

⟨ direct action id ⟩ ⟨ special code ⟩ .

The meaning of the metalinguistic symbol is:

⟨ direct action id ⟩. This is a mnemonic

representing the desired direct action as specified in the language symbol definition program.

⟨ special code ⟩. This may be any set of inputs in any syntax that has been provided for in the translator. There is no set syntax for this part of direct statements. Any statement may be preceded by a ⟨ label ⟩. A ⟨ label ⟩ is an integer used to identify a statement, and must be used in all references to that statement. The last statement on /PROG/ must be an "END" statement. The format is:

END (CR) .

b. Output

The translator produces two drum files as output:

(1) /TREE/--This file contains the translated statements and other information required for execution of the program by the interpreter. This information is:

(a) Translated source code

(b) Packing formats

(c) Number of direct actions

(d) Number of statements.

(2) /LABS/--This is a file that enables the untranslator to reconstruct the labels used in a source file when "untranslating" a translated file.

c. Implementation Information

The translator first reads in the symbol table and miscellaneous information from the file /ITAB/, and then uses this information in translating /PROG/. Each statement is translated roughly as follows:

(1) Read in the first symbol on the file. If it is an integer, then consider it a label, and store it in a list of labels so that it corresponds to this statement. Read the next symbol of the record.

(2) If the current symbol is not alphanumeric, flag an error and abandon this statement. Otherwise look up in symbol table. If not there, flag an error. Otherwise replace symbol by value found in symbol table.

(3) If the symbol value represents a direct action, call routine "ISPEC" to process the rest of the statement. Otherwise scan for the remaining four fields of a test statement, and flag an error if any incorrect fields are detected.

(4) Clear input buffer to start of next line.

When all statements have been translated, a pass is made through the translated code, and all labels are replaced by the correct addresses. Any undefined labels are flagged at this point. The translated code is then written onto the file /TREE/.

4. The Interpreter

The interpreter executes the instructions produced by the translator. While there are only two basic statement types, the interpreter performs three basic functions: test execution, directive execution, and indirect action execution. Indirect actions are actions that are specified by a test statement, and are executed when a node is reached via a given branch.

The path of control through the decision tree is not governed by the binary decisions made at each node, but rather by the confidences generated by executing a test. Each test produces two nodes when executed. Associated with each of these nodes is a confidence, which is a function of the confidences of previous nodes along the path to the test and the results of the test just executed. These nodes and their confidences are placed on a list of open nodes. This list is then scanned, and the node with the highest confidence is removed and chosen to be the next node to be processed. Control continues in this manner until a terminal node is reached.

Associated with each open node is a trace of all nodes parsed on the path to that node. This trace contains the values of all registers, the confidence, the depth of the node in the tree, and other miscellaneous information. During execution this information can be printed so that the user can monitor the operation of the tree.

There are three places in the interpreter where the user must provide code. These correspond to directives, indirect actions, and tests as specified in the language. The extent of this code will normally be a subroutine, and a corresponding subroutine call. A selection of utility subroutines are available, which should substantially reduce the amount of coding required. A user of the interpreter should know the following technical information.

a. Input Data

The interpreter requires four drum files as input:

(1) /TREE/--This file contains the 'object code' and packing formats produced by the translator.

(2) /ICIR/--This file, which contains the rough coordinates of a circle, is needed by the spur-finding routine. The file /ICIR/ is binary, and should not be confused with the file /ICIRCLE/, which contains the same information in a symbolic format. The program CVICIRCLE will produce /ICIR/ from /ICIRCLE/.

(3) Vertical Line File--This file contains lists of vertical lines that correspond to the pictures on the gray-scale picture file (4). This file is currently produced by executing two programs, VERTGETTER and CVVERT. VERTGETTER produces a symbolic file /VL/ from a gray-scale picture file. /VL/ contains the vertical lines found in the frames of the gray-scale picture file. CVVERT then converts /VL/ to a packed binary

form, and writes it onto the vertical line file. The actual name of the vertical line file is entered as data when the interpreter is executed.

(4) Gray-Scale Picture File--This file contains the raw gray-scale data for one or more television pictures. The name of this file is entered as data when the interpreter is executed.

The interpreter also requires some manual input from the teletype. The requests for information and possible responses are:

(1) PICTURE NO.--Respond with ⟨ integer ⟩ ⟨,⟩ indicating which picture on the gray-scale picture file you wish to process.

(2) GRAY SCALE FILE NAME= --Respond with the name of appropriate gray-scale picture file name, followed by a period (command recognizes file name).

(3) VERTICAL LINE FILE NAME= --Respond with the name of appropriate vertical line file followed by a period. This must correspond to the gray-scale picture file name used in (2). This name is command recognized.

(4) OPTIONS--This puts the interpreter into a loop that allows the user to specify the printout and displays he wishes to see during the processing of the picture. All options have a default setting, which is not modified through successive runs of the interpreter so long as it is not reloaded. The possible options are:

(a) STEP (Command recognizes on "S." Default = 0)--This causes the interpreter to wait for a carriage return before each new display is shown. This option implies the DISPLAY option.

(b) DISPLAY (Command recognizes on "D." Default = 1)--This causes certain steps of the processing of the picture to be displayed on the scope. The steps displayed are controlled by the program being executed.

(c) TRACE (Command recognizes on "T." Default = 0)--This causes the address of each node transferred to be printed on the teletype.

(d) CONFIDENCES (Command recognizes on "C." Default = 0)--This causes the confidence and level of each node processed to be printed, along with the confidence of the test at that node. This option implies the trace option.

(e) GO (Command recognizes on "G." Default = not apply)--This causes the picture processing to begin, and exits the option loop.

(f) NIL (Command recognizes on "N." Default = not apply)--This causes the effect of any option following it to be negated. For example, NIL STEP turns the step option off.

b. Output

The output from the interpreter is totally in the form of printed messages on the teletype. These messages are initiated by the executing program, or by one of the options as specified above. In addition, certain self-explanatory diagnostic and run-time error messages may be printed in the output.

c. Implementation Information

The interpreter first reads all of the necessary files for a run, and calls routines to initialize parameters. After all parameters are set, and any options desired are specified, the control section of the interpreter enters a program execution loop. The flow of this loop is roughly:

(1) Scan open node list and retrieve address of next instruction to be processed (this is the address of the node with highest confidence).

(2) Increment the level counter.

(3) Execute any indirect action required at the node.

(4) Fetch and decode instruction at node.

(5) If instruction is a direct action, execute action, increment instruction counter, and go to (4).

(6) Execute test as indicated by instruction.

(7) Based on the results of the test, enter two nodes into the open node list, one for the true confidence and branch, and one for the false confidence and branch.

The direct actions are executed by a routine DIRECT, the indirect actions are executed by a routine ACTION and the tests are executed by a routine ITEST. Each of these routines calls subroutines that execute the particular functions desired. Details concerning modification are provided at the front of each of these routines. When the program execution loop is completed, all parameters (with the possible exception of file names) are reinitialized.

5. The Untranslator

This program reads a code file produced by the translator, and lists it on the teletype in its original form, save for comments. It types an identification number by each statement, corresponding to the identification printed at each node by the interpreter. The user is required to provide the code for printing directive statements, and should know the following information.

a. Input Data

The untranslation program requires three drum files as input:

(1) /TREE/--This file contains the program "object code" and other miscellaneous information. It is the main output of the translator.

(2) /ITAB/--This is the symbol table file produced by the language symbol definition program.

(3) /LABS/--This is a file containing label information pertaining to the "object code" on file /TREE/. It is created as an output of the translator.

b. Output

The untranslation program produces a listing of the object program on /TREE/. The listing has one statement per line, and each statement is preceded by its address in brackets. This address corresponds to the address printed by the trace option in the interpreter.

c. Implementation Information

The untranslator reads in the symbol table (from /ITAB/) and label information (from /LABS/). The object file is then read, and each statement is listed until an END is encountered. If a statement is a test statement, it is listed in a predetermined format. Otherwise a routine SPECIAL is called, and SPECIAL lists the statement in any format desired.

6. A Decision Tree for Scene Analysis

To illustrate the way in which a decision tree can be programmed using this software, we give here the formal definitions of the direct actions, indirect actions, and tests employed in an actual implementation, and a listing of a complete decision-tree program.

a. Direct Actions

⟨ move ⟩ :=MOV ⟨ integer ⟩ : ⟨ register ⟩ ← ⟨ register ⟩|
⟨ move ⟩ ; ⟨ integer ⟩ : ⟨ register ⟩ ← ⟨ register ⟩

Semantics: Move moves ⟨ integer ⟩ words from ⟨ register ⟩ #1 to ⟨ register ⟩ #2.

⟨ branch ⟩ :=BRU ⟨ label ⟩

Semantics: Branch transfers control to the node designated by ⟨ label ⟩

⟨ follow ⟩ :=FOL ⟨ register ⟩

Semantics: Follow follows a line indicated by the contents of four cells (X1,Y1,X2,Y2) beginning with the cell designated by ⟨ register ⟩. It returns the coordinate (X,Y) of the farthest point reached in R1, R2, and the confidence in R3.

⟨ getspurs ⟩ :=GSP ⟨ register ⟩

Semantics: Get spurs finds all of the spurs at the point designated by two cells at ⟨ register ⟩, except for those in the direction of the point designated by the two cells (X2,Y2) at ⟨ register ⟩ +2,+3. The line pointer for the list of spurs is returned in R1,R2.

b. Indirect Actions

⟨ follow ⟩ :=FOL

Semantics: Follows the line designated by (R1;X1,R2;Y1, R3;X2,R4;Y2) and returns the endpoint of the new line in (R5;X,R6;Y). The confidence is returned in R7.

⟨getspurs ⟩ :=GSP

Semantics: Works as does the direct action GSP, except the points (X1,Y1),(X2,Y2) are taken from R1,R2,R3,R4.

c. Tests

General Form: ⟨ test ⟩ := ⟨ mnemonic ⟩ ⟨ + node ⟩ ⟨ indirect action ⟩ ⟨ - node ⟩ ⟨ indirect action ⟩

Semantics: The test indicated by the ⟨ mnemonic ⟩ is executed, and the confidence for the ⟨ + node ⟩ is computed using the confidence of that test,

and the confidence of the node at which the test is being performed (the current node). The confidence of the ⟨ - node ⟩ is computed from 100-confidence and the confidence of the current node. The nodes their confidences, and their respective actions are then stored into the table of open nodes. They are linked to a trace of the preceding nodes en route to the current node, and the current node is entered as the most recent entry of that trace. Information currently stored in the trace is the contents of R1 to R8 and the confidence of the test.

VLR (Test #1) (Vertical Line Remove): Finds the most confident vertical line from the global list of vertical lines, and removes it from this list. Returns the coordinates of the lower endpoint in R1,R2. and the upper endpoint in R3,R4 unless the confidence is 0, in which case R1-4 are not altered. The confidence is returned internally in ICONWD

Input: none

Output: confidence = 0: none

confidence > 0: R1,R2,R3,R4

POP (Test #2) (Point On Picture): Returns the confidence of the point designated by (R1,R2) being on the picture. This confidence is returned internally in ICONWD.

Input: R1,R2

Output: none

SPR (Test #3) (Spur Remove): Finds the most confident spur in the list indicated by the line pointer in R1,R2. Returns the coordinates of this spur in R1,R2,R3,R4 unless the confidence = 0. The confidence is returned internally in ICONWD. The spur found is removed from the list.

Input: R1,R2

Output: confidence = 0: none

confidence > 0: R1,R2,R3,R4

BAS (Test #4) (Baseboard): Tests to see if spurs in list indicated by line pointer in R1,R2 are part of the baseboard, and returns the confidence internally in ICONWD.

Input: R1,R2

Output: none

DSP (Test #5) (Directed Spur): Looks for spurs in the direction of point (R3,R4) from point (R1,R2). Returns the coordinates of the closest one in R1,R2,R3,R4, and the angle of deviation from the desired spur in R5 (rounded to nearest degree). The confidence of this spur is returned internally in ICONWD.

Input: R1,R2,R3,R4

Output: R1,R2,R3,R4,R5

CON (Test #6) (Connect): Trys to connect the 2 points (R1,R2), (R3,R4). Returns the coordinates of the resulting line in (R1,R2,R3,R4). The confidence is returned internally in ICONWD.

Input: R1,R2,R3,R4

Output: R1,R2,R3,R4

EQL (Test #7) (Equal Lines): Tests two lines for equality, and returns the confidence of equality internally in ICONWD, i.e.,

$$[(R1,R2),(R3,R4)] = [(R5,R6),(R7,R8)]$$

Input: R1,R2,R3,R4,R5,R6,R7,R8

Output: none

WDG (Test #8) (Wedge Test): Tests to see if two lines are part of a wedge, and returns the confidence of the test

internally in ICONWD. The lines tested are in R1,R2,R3,R4, R5,R6,R7,R8.

Input: R1,R2,R3,R4,R5,R6,R7,R8

Output: none

SYMBOLIC DECISION TREE FOR SCENE ANALYSIS

LABEL	OPER	T.BR	T.ACT	F.BR	F.ACT	COMMENTS
	EVT	2	0	999	101	NO VERTICALS
2	MOV	4:1(1)-1				
	POP	3	0	19	0	
3	ESP	4	0	999	102	WALL OR UNLIKELY
4	BAS	5	0	999	103	DOOR
5	EVT	7	0	310	0	
7	POP	8	0	6	0	
8	MOV	4:2(1)-1; 2:3-1; 2:1-1(1)				
	DSP	9	0	5	0	
6	MOV	4:2(1)-1; 2:1-1(1)				
	CON	25	0	5	0	
9	CON	10	0	5	0	
10	OHZ	101	0	11	0	
11	MOV	4:4(1)-1(1)				
	OHZ	12	0	999	104	OBJECT AT
101	MOV	4:4(1)-1(1); 4:1(5)-2(1)				VL AND VO
	BRU	13				
12	MOV	4:4(1)-2(1); 4:1(5)-1(1)				VL AND VO
	BRU	13				
13	OVT	14	0	131	0	
131	MOV	4:1-4(1); 4:5-2(1)				
	EQL	132	0	133	0	
133	MOV	2:1-2(1)				
	OHZ	12	0	134	0	
134	MOV	4:4(1)-1(1); 4:1(5)-2(1)				
	BRU	50				
132	MOV	4:4(1)-2(1); 4:1(5)-1(1)				VL AND VO
	BRU	50				
14	MOV	4:5(1)-1				SAVE OFF V
	MOV	2:3-1; 2:1-4(1)				
	DSP	15	0	13	0	
15	CON	16	0	13	0	
16	MOV	4:3(1)-5(1); 2:1-3(3)				
	POP	17	0	999	105	OBJECT AT
17	MOV	2:1-1(7)				THIS IS VO
	POP	18	0	999	105	OBJECT AT
18	MOV	2:3-3(3)				
	CON	999	106	999	107	WEDGE ON SIDE VS CUBE
19	MOV	2:1-1(3)				
	POP	20	0	999	108	DOOR; DRAPES; TOO CLOSE
20	ESP	21	0	999	109	GIVE UP ON THIS ONE
21	OVT	22	0	501	1	
22	MOV	4:2(1)-1; 2:1-3				
	POP	23	0	21	0	
23	MOV	2:1-1(3)				
	DSP	24	0	21	0	
24	CON	25	0	21	0	
25	MOV	2:1-1(3)				
	EHZ	27	0	26	0	
26	MOV	2:1-2(3)				
	EHZ	261	0	999	110	OBJECT AT
27	MOV	2:1-2(3)				
	EHZ	999	111	262	0	CUBE VS WEDGE ON SIDE
261	MOV	4:4(1)-2(1)				
	BRU	26				

```
282   MOV   4:4(1)-1(1)
28    WDG   999    112    999    113     WEDGE VS OBJECT
310   MOV   2:1-1(3)
      POP   311    0      31     0
31    MOV   2:1-1(1)
      EHZ   32     1      999    112     UNLIKELY SITUATION
32    POP   33     0      999    113     OBJECT AT
33    MOV   2:5(1)-1
      ESP   34     1      999    114     UNLIKELY SITUATION
34    WDG   999    115    999    116     WEDGE VS UNLIKELY SIT.
311   ESP   501    1      999    117     UNLIKELY SITUATION
501   BRU   506
506   POP   502    0      999    118     CUBE VS WEDGE
502   WDG   999    119    503    0       WEDGE
503   MOV   2:1-1(3)
      BRU   311
50    MOV   2:1-4(3)
      POP   51     0      52     0
51    ESP   57     FOL    53     0
52    FOL   6(1)
      POP   62     0      999    120     OBJECT AT
62    ESP   63     0      999    121     WEDGE AT
63    MOV   4:1-2(1); 4:5-4(1)
      EQL   64     0      999    122     UNLIKELY SITUATION
64    MOV   2:1-1(1)
      ESP   56     0      999    123     UNLIKELY SITUATION
53    FOL   6(1)
      POP   54     0      999    124     OBJECT OFF PICTURE
54    MOV   4:1-2(1); 4:5-4(1)
      EQL   55     0      999    125     UNLIKELY SITUATION
55    MOV   2:1-1(1)
      ESP   56     0      999    126     UNLIKELY SITUATION
56    MOV   4:4(1)-1(1)
      BRU   50
57    WDG   999    127    59     0       WEDGE AT
59    MOV   4:1-4(1); 4:5-1(1)
      EQL   60     GSP    999    128     CUBE AT
60    ESP   56     0      999    128     CUBE AT
999   END
```

Appendix E

EXPERIMENTS WITH COLOR PICTURES

Appendix E

EXPERIMENTS WITH COLOR PICTURES

To test the ability of our television input system to discriminate on the basis of hue and saturation, rather than just on the basis of intensity, a series of 25 test pictures were taken of 25 colored felts. Three pictures were taken of each felt through three different Wratten filters: No. 29 (red), No. 58 (green), and No. 47 (blue). To compensate roughly for the spectral characteristics of our 8507 vidicon, a 50 percent neutral-density filter was used in conjunction with the green filter. As it turned out, this provided the desired compensation to within the accuracy of the 16-level intensity quantization used.

To a first degree of approximation, the relation between the illumination E of the vidicon faceplate and the output V of the A/D converter is

$$V = V_o + CE^{\gamma}$$

where V_o is an offset constant for the A/D converter, C is a vidicon transmission coefficient that varies with wavelength, and γ is a constant characteristic of the vidicon. Given these constants, one can compute the incident illumination for each color from the corresponding values in the digitized pictures.

The use of a three-color system and the idea of using intensity, hue, and saturation as a basis for extracting information from pictures are based, of course, on well-known facts about the psychology of human vision. In principle, one could use the entire spectral distribution to obtain even more information about the environment. However, this kind of information is both difficult to measure and difficult to interpret. With a three-color system, acceptable signal-to-noise ratios can be obtained under normal illumination. Furthermore, the procedures using

unified trichromatic coefficients to compute dominant wavelength and purity (the physical quantities corresponding roughly to the sensation attributes of hue and saturation) have long been well established and are easy to interpret.

Let E_r, E_g, and E_b be the illumination values for the red, green, and blue pictures, respectively. Then the unified trichromatic coefficients are given by*

$$x = \frac{E_r}{E_r + E_g + E_b}$$

$$y = \frac{E_g}{E_r + E_g + E_b}$$

$$z = \frac{E_b}{E_r + E_g + E_b}$$

Since x, y, and z sum to one, any color becomes represented as a point in the x-y plane within the color triangle. Ideally, white is located at $x_w = 1/3$, $y_w = 1/3$, and pure spectral colors appear along a locus that hugs the boundaries of the triangle. If a line drawn from (x_w, y_w) through (x, y) intersects the spectral locus for light of wavelength λ at (x_λ, y_λ), λ defines the dominant wavelength, and the colorimetric purity p is given by

$$p = \frac{1}{1 + \frac{y_w}{y_\lambda}\left[\frac{x_\lambda - x}{x - x_w}\right]}$$

Thus, the purity is zero for white light, one for a pure spectral color, and varies uniformly from zero to one along the line from (x_w, y_w) to (x_λ, y_λ).

---

*Strictly speaking, these and the following formulas are correct only if the combined transfer function for the vidicon and the filters yields the C.I.E. standard curves. See P. Moon, The Scientific Basis of Illuminating Engineering, Chap. 13 (Dover, New York, 1961).

In our experiments, the accuracy limitations imposed by 16-level quantization made the accurate calculation of λ and p seem unwarranted, although either can be roughly estimated from our data. Even the calculation of x and y proved to be quite sensitive to quantization. This was particularly true at low intensities, where a change of one level could cause a fairly large change in the position of the point in the x-y plane.

Each colored felt was viewed both horizontally oriented under bright overhead illumination, and vertically oriented where it received more illumination by reflection from the floor and walls. (Had this merely reduced the intensity of the illumination, it should have had no effect on the x-y coordinates.) The quantized intensity values over 176-cell areas in the light and dark regions were averaged to give red, green, and blue values for V, which were then used to compute E_r, E_g, and E_b using $V_o = 0.4$, $C = 1$, and $\gamma = 0.67$. The corresponding points in the color triangle are shown in Figure E-1 as vectors from the point for the light region to the point for the dark region.

Inspection of Figure E-1 shows that hue (dominant wavelength) discrimination is fairly good. The spectrum from red to blue is spread out counterclockwise around the point for white, as it should be, and such colors as red, orange, yellow, green, blue, and purple can be easily distinguished. The various shades of pink indicate that some discrimination on the basis of saturation (purity) is also possible, although there is clearly a significant danger of confusing the hues of unsaturated colors. It is clear that this danger will also be present whenever the illumination levels are low. Thus, the primary conclusion of these experiments is that our present television system can provide the information needed to discriminate between at least six different hues, but that reliable operation requires high saturation, good illumination, and a moderately large visible region of the picture over which to average.

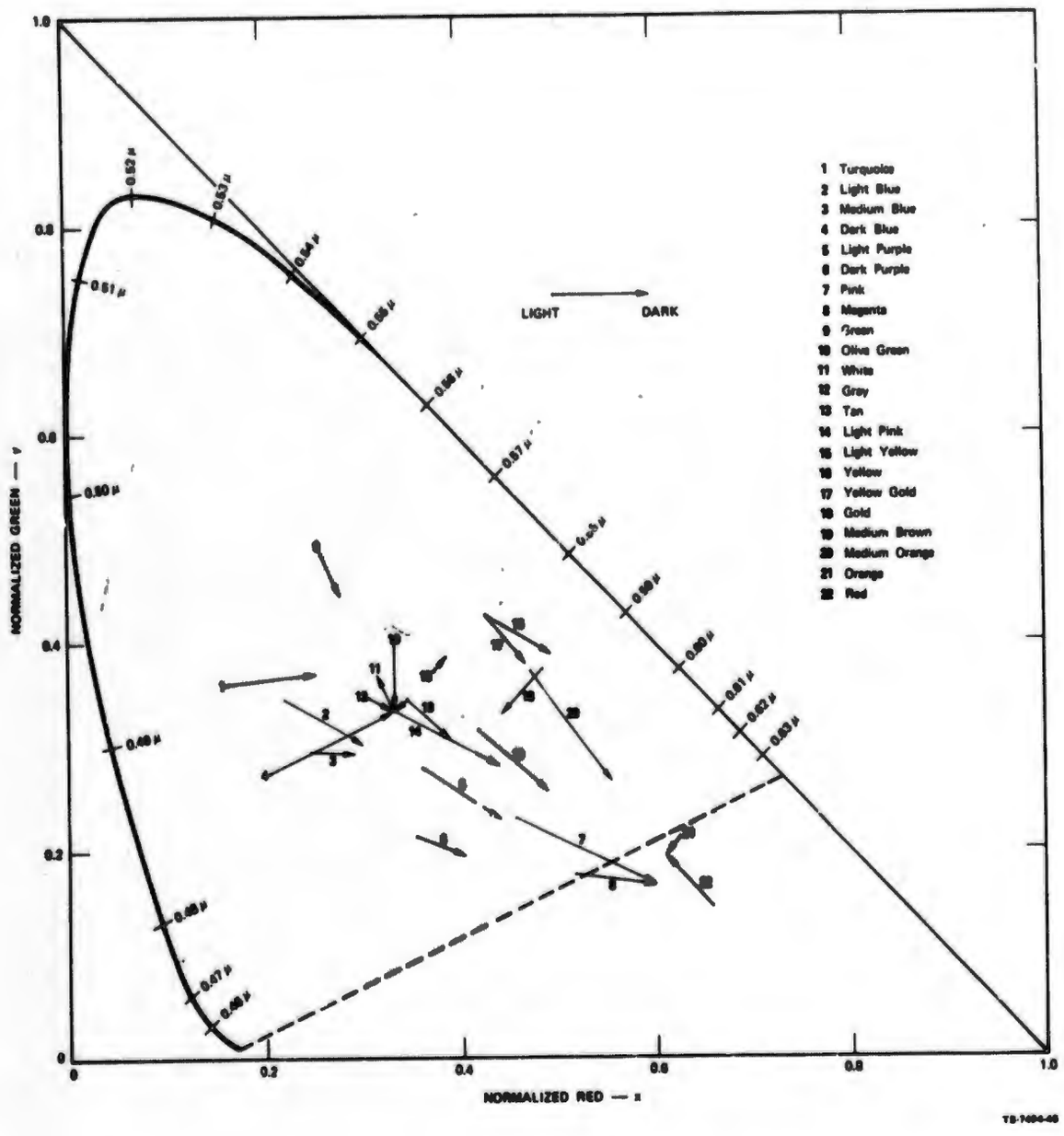


FIGURE E-1 EXPERIMENTAL RESULTS PLOTTED ON A CHROMATICITY DIAGRAM

Appendix F

PUBLICATIONS AND PAPERS PREPARED UNDER CONTRACT F30602-69-C-0056

BLANK PAGE

Appendix F

PUBLICATIONS AND PAPERS PREPARED UNDER CONTRACT F30602-69-C-0056*

Papers

L. Stephen Coles, "Talking with a Robot in English," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C. (7-9 May 1969).

Cordell Green, "Application of Theorem Proving to Problem Solving," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C. (7-9 May 1969).

N. Nilsson, "A Mobile Automaton: An Application of Artificial Intelligence Techniques," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C. (7-9 May 1969).

R. Yates and B. Raphael, "Resolution in Graphs" (in preparation).

Report

Nils J. Nilsson, "Research on Intelligent Automata," First Interim Report, covering the period 7 August 1968 to 28 February 1969, SRI Project 7494, Stanford Research Institute, Menlo Park, California (February 1969).

Technical Notes

Peter J. Hart, "Searching Probabilistic Decision Trees," Artificial Intelligence Group Tech. Note 2 (February 1969).

Richard O. Duda and Peter J. Hart, "Perspective Transformations," Artificial Intelligence Group Tech. Note 3 (February 1969).

C. Cordell Green, "Application of Theorem Proving to Problem Solving," Artificial Intelligence Group Tech. Note 4 (March 1969).

David A. Huffman, "Logical Analysis of Pictures of Polyhedra," Artificial Intelligence Group Tech. Note 6 (May 1969).

Claude Fennema and Claude Brice, "A Region-Oriented Data Structure," Artificial Intelligence Group Tech. Note 7 (May 1969).

---

*SRI Project 7494.

L. Stephen Coles and C. Cordell Green, "Chemistry Question Answering," Artificial Intelligence Group Tech. Note 9 (June 1969).

John H. Munson, "The N-Tuple Storage System," Artificial Intelligence Group Tech. Note 10 (August 1969; addendum October 1969).

John H. Munson, "A LISP-to-FORTRAN Interface for the PDP-10," Artificial Intelligence Group Tech. Note 12 (September 1969; addendum October 1969).

Bertram Raphael, "The Relevance of Robot Research to Artificial Intelligence," Artificial Intelligence Group Tech. Note 13 (May 1969). [To appear in Proc. 4th Systems Symposium, Formal Systems and Non-Numerical Problem Solving by Computers, Case Western Reserve University, Cleveland, Ohio (19-20 November 1969).]

Thomas D. Garvey, "User's Guide to the QA3.5 Question-Answering System," Artificial Intelligence Group Tech. Note 15 (October 1969).

Claude Fennema and Claude Brice, "Scene Analysis of Pictures using Regions," Artificial Intelligence Group Tech. Note 17 (November 1969) [being submitted to Artificial Intelligence].

REFERENCES

1. C. A. Rosen et al., "Application of Intelligent Automata to Reconnaissance," Final Report, Contract AF 30(602)-4147, SRI Project 5953, Stanford Research Institute, Menlo Park, California (December 1968).

2. C. Rosen and N. Nilsson (eds.), "Application of Intelligent Automata to Reconnaissance," Third Interim Report, Contract AF 30(602)-4147, SRI Project 5953, Stanford Research Institute, Menlo Park, California (December 1967).

3. N. Nilsson (ed.), "Research on Intelligent Automata," First Interim Report, Contract F30602-69-C-0056, SRI Project 7494, Stanford Research Institute, Menlo Park, California (February 1969).

4. N. Nilsson, "A Mobile Automaton: An Application of Artificial Intelligence Techniques," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C. (May 1969).

5. Cordell Green, "Application of Theorem Proving to Problem Solving," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C. (May 1969).

6. L. Stephen Coles, "Talking with a Robot in English," Proc. Int'l. Joint Conf. on Artificial Intelligence, Washington, D.C., 7-9 May 1969.

7. B. Raphael, "The Relevance of Robot Research," Artificial Intelligence Group Technical Note 13, Contract F30602-69-C-0056, SRI Project 7494 (May 1969). To appear in Proc. of the Fourth Systems Symposium, Formal Systems and Non-Numerical Problem Solving by Computers, Case Western Reserve University, Cleveland, Ohio, 19-20 November 1968.

8. R. Yates and B. Raphael, "Resolution Graphs" (in preparation).

9. C. Brice and C. Fennema, "Scene Analysis of Pictures Using Regions" (in preparation).

10. J. McCarthy, L. D. Earnest, D. R. Reddi, and P. J. Vicens, "A Computer with Hands, Eyes, and Ears," Proc. Fall Joint Computer Conf., Vol. 33, Part 1, pp. 329-338 (Thompson Book Company, Washington, D.C., 1968).

11. L. G. Roberts, "Machine Perception of Three-Dimensional Solids," in Optical and Electro-Optical Information Processing, J. L. Tippett et al. (eds.), pp. 159-197 (MIT Press, Cambridge, Massachusetts, 1965).

12. G. E. Forsen, "Preprocessing Visual Data with an Automaton Eye," in Pictorial Pattern Recognition, G. C. Chang et al. (ed.), pp. 471-502 (Thompson Book Company, Washington, D.C., 1968).

13. A. Guzman, "Decomposition of a Visual Scene into Three-Dimensional Bodies," Proc. Fall Joint Computer Conf., Vol. 33, Part 1, pp. 291-304 (Thompson Book Company, Washington, D.C., 1968).

14. A. Guzman, "Object Recognition: Discovering the Parallepipeds in a Visual Scene," Proc. 2nd Hawaii Int. Conf. Sys. Sci., pp. 479-482 (January 1969).

15. P. E. Hart and R. O. Duda, "Perspective Transformations," SRI Artificial Intelligence Group Tech. Note 3, Contract F30602-69-C-0056, SRI Project 7494, Stanford Research Institute, Menlo Park, California (February 1969).

16. C. Fennema and C. Brice, "A Region-Oriented Data Structure," SRI Artificial Intelligence Group Technical Note 7, Contract F30602-69-C-0056, SRI Project 7494, Stanford Research Institute, Menlo Park, California (May 1969).

17. J. A. Robinson, "Mechanizing Higher-Order Logic," in Machine Intelligence, Michie and Meltzer (eds.) (Edinburgh University Press, Scotland, 1969).

18. W. Ash and E. Sibley, "TRAMP: An Interpretive Associative Processor with Deductive Capabilities," Proc. 1968 ACM Conference, pp. 143-156 (1968).

19. J. Feldman and P. Rovner, "An ALGOL-Based Associative Language," Comm. ACM, Vol. 12, No. 8. pp. 439-449 (August 1969).

UNCLASSIFIED
Security Classification

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 Ave.
Menlo Park, CA 94025

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
N/A

3. REPORT TITLE
APPLICATION OF INTELLIGENT AUTOMATA TO RECONNAISSANCE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Report 7 August 1968 to 7 November 1969

5. AUTHOR(S) *(First name, middle initial, last name)*
L. Stephen Coles, Charles A. Rosen, John H. Munson
Bertram Raphael, Thomas D. Garvey
Richard O. Duda, Robert A. Yates

6. REPORT DATE
November 1969

7a. TOTAL NO. OF PAGES
164

7b. NO. OF REFS
19

8a. CONTRACT OR GRANT NO.
F30602-69-C-0056

b. ~~XXXXXXXX~~
ARPA Order No. 1058, Amend. #1

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
SRI Project 7494

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
RADC-TR-69-464

10. DISTRIBUTION STATEMENT
This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMIDD), GAFB, N.Y. 13440

11. SUPPLEMENTARY NOTES
Monitored by
Patricia M. Langendorf 315-330-2621
RADC (EMIDD), GAFB, NY 13440

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
Washington, DC 20301

13. ABSTRACT

A 14-month project of research in the application of techniques of artificial intelligence to the control of a mobile automaton in a realistic environment is described. The main emphasis is on experimentation with a previously-developed system of hardware and software, and on research in several related areas of artificial intelligence where new efforts have been necessary to increase the capabilities of the automaton. Major areas discussed include the use of formal theorem-proving techniques of first-order logic in solving problems for the automaton; symbolic information structures for modeling the automaton's environment; results in visual scene analysis, including a decision-tree approach and the use of regional as well as local analysis; and an outline for the design of a problem-solving system based on higher-order logic.

DD FORM 1 NOV 68 1473

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Artificial intelligence						
Mobile automata						
Theorem Proving						
First-order predicate calculus						
Computer programming						
System Programming						
Natural - language processing						
Modeling						
Visual scene processing						
Television processing						
project geometry						
Problem-solving systems						
Question answering						
Robot systems						
Reconnaissance systems						

AFLC—Griffiss AFB NY 23 Apr 70-63