Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots

by

Erann Gat

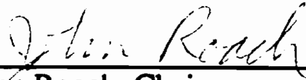
A Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:



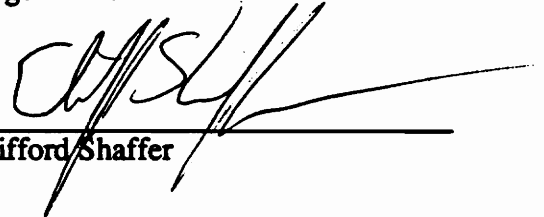
John Roach, Chairman

David P. Miller, Advisor

Rodney Brooks

Roger Ehrich

Adrienne Bloss



Clifford Shaffer

19 April, 1991
Blacksburg, Virginia

c.2

LD
5655
V856
1991
G39
c.2

Copyright © by Erann Gat, 1991. All rights reserved.

Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots

by

Erann Gat

Committee Chairman: John Roach
Major Advisor: David P. Miller

(ABSTRACT)

This dissertation demonstrates that effective control of autonomous mobile robots in real-world environments can be achieved by combining reactive and deliberative components into an integrated architecture. The reactive component allows the robot to respond to contingencies in real time. Deliberation allows the robot to make effective predictions about the world. By using different computational mechanisms for the reactive and deliberative components, much existing deliberative technology can be effectively incorporated into a mobile robot control system.

The dissertation describes the design and implementation of a reactive control system for an autonomous mobile robot which is explicitly designed to interface to a deliberative component. A programming language called ALFA is developed to program this system.

The design of a control architecture which incorporates this reactive system is also described. The architecture is heterogeneous and asynchronous, that is, it consists of components which are structured differently from one another, and which operate in parallel. This prevents slow deliberative computations from adversely affecting the response time of the overall system. The architecture produces behavior which is reliable and goal-directed, yet reactive to contingencies, in the face of noise, limited computational resources, and an unpredictable environment.

The system described in this dissertation has been used to control three real robots and a simulated robot performing a variety of tasks in real-world and simulated real-world environments. A general design methodology based upon bottom-up hierarchical decomposition is demonstrated. The methodology is based on the principle of cognizant failure, that is, that low-level activities should be designed in a way as to detect failures and state transitions at high levels of abstraction. Furthermore, the results of deliberative computations should be used to guide the robot's actions, but not to control those actions directly.

Acknowledgements

First and foremost I would like to thank my advisor, David Miller, for five years of advice and friendship. Dave gave me the freedom to pursue my own ideas, and somehow managed to bring me back to reality when they got too outlandish (which was not an infrequent occurrence). Whatever merit this thesis may possess, it is largely due to Dave's efforts. All the bogosity is my fault.

Second, I would like to thank Rodney Brooks, both for allowing me to spend the summer of 1990 working in his lab, and for taking time out of his fantastically busy schedule to serve on my committee. I am also grateful to the mobot lab gang for making that summer immensely enjoyable and productive. Special thanks go to Maja Mataric for letting me be her surrogate that summer and loaning me her robot, Anita Flynn for her unfailing encouragement, Cindy Ferrell and Chuck Rosenberg for hanging in there for three long days as we tried to get Atilla to walk (We almost made it!) and Colin Angle for building Tooth, the robot, and being a really cool dude.

Third, I would like to thank the other members of my committee. John Roach served as chairman. Roger Ehrich, Adrienne Bloss, and Cliff Shaffer all made time to serve on short notice. I would also like to thank the members of my preliminary committee who were not able to be on my final committee: Morton Nadler, Pat Bixler, and H.R. Vanlandingham.

I would like to extend special thanks to Jim Firby. In the short time I have had the privilege of knowing him, Jim has shaped my thoughts more than any other person besides my advisor. Jim provided invaluable comments on an early draft of this thesis, and wrote the simulator used in some of the experiments. He also taught me a little about programming.

I am also extremely grateful to Marc Slack, Paul Viola, Rajiv Desai and Steve Chien for reading and commenting on early drafts of my thesis. Special thanks to Marc for showing me that finishing is possible, Paul for many interesting and educational conversations, Rajiv for teaching me a bit about politics, and Steve for moral support and comic relief.

Many people at JPL provided support and assistance for the experiments I performed on Robbie, the planetary rover testbed. Special thanks go to John Loch and Larry Matthies for their help with the Robbie experiments, and Greg Dewitt for being a great bunch of boulders. Thanks to David Atkinson and Brian Wilcox for making it all possible.

This thesis would not have been possible without a great deal of advice from a great many people. I cannot possibly list them all here, and I apologize at the outset for any I might have slighted. That said, I would like to thank Maja Mataric, Ian Horswill, Leslie Kaelbling, Stan Rosenschein, Marcel Schoppers, Nils Nilsson, Robin Murphy, Monett Soldo, Lynn Stein, and Boris Katz.

Thanks to Heather Bryden for inviting me to come to Hoboken. None of this would have been possible without you, Heather.

Thanks to Agness Chandler of the Virginia Tech Computer Science department whose help was invaluable when it came time to arrange my thesis defense by remote control.

Thanks to my family for their love and support. I would especially like to thank my parents who instilled in me very young a love of learning. Mom, Dad, thanks for all the books.

Finally, I would like to thank my wife, Alison. Marrying her was the smartest thing I ever did. To her, and everyone else who supported me over the past five years: I'm sorry I was such a pain in the ass.

This research was supported in part by a graduate fellowship from the Office of Naval Research administered by the American Society for Engineering Education, and was conducted in part at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology under the sponsorship of Professor Rodney Brooks, and in part at the California Institute of Technology Jet Propulsion Laboratory under a contract with the National Aeronautics and Space Administration.

Apologia: A Note on the use of the First Person

In this thesis I use the first person. This has elicited such a volume of criticism that a word of explanation is in order.

There are three distinct ways in which the first person can be used in a technical document. The first is while reporting a method or result, e.g., "I programmed the robot to drive around rocks." The second is when making self-referential statements about the writing itself, e.g., "I will describe the process in three steps." The third is to illustrate a point with an example, e.g., "When I drive through downtown Hoboken I need to look for oncoming cars."

In this thesis, I use the first person only in the third way and, occasionally, the second. I use the first person to describe my intentions as a writer only when it is not appropriate to ascribe those intentions to "this chapter" or "this thesis" (as in this sentence). I use the first person in illustrative examples because examples are, by their very nature, informal. Replacing the first person in such cases usually involves replacing "I" with "one" or the use of the passive voice. The result is unnecessary stuffiness. Formality is fine, but it should not be allowed to interfere with clarity.

'If a cat can kill a rat in a minute, how many would be needed to kill it in the thousandth part of a second?' The *mathematical* answer, of course, is '60,000,' and no doubt less than this would *not* suffice; but would 60,000 suffice? I doubt it very much. I fancy that at least 50,000 of the cats would never even see the rat, or have any idea of what was going on.

Or take this: 'If a cat can kill a rat in a minute, how long would it be killing 60,000 rats?' Ah, how long, indeed! My private opinion is that the rats would kill the cat.

--Lewis Carroll

Down this street, two lights and turn left. You can't miss it.

--An anonymous gas station attendant
in Hoboken, New Jersey

Ignore Broadway. It's weird.

--Heather Bryden

Table of Contents

Chapter 1: Introduction 1
1.1 Driving to Hoboken 2
1.2 Issues in Real-World Navigation 5
1.2.1 Local Sensor Data vs. Internal State 6
1.2.2 Dealing with Sensor Noise 7
1.2.3 Real-Time Response 7
1.2.4 Dealing with Failure 8
1.2.5 The Role of Plans 9
1.3 Actions vs. Activities 10
1.3.1 What are Actions? 10
1.3.2 Activities and Decisions 12
1.3.3 Levels of Activity 13
1.3.3.1 Commitment 14
1.3.3.2 Computation Time 15
1.3.3.3 The Role of Internal State 16
1.3.3.4 Levels of Abstraction 17
1.4 An Architecture for Autonomous Navigation 18
1.4.1 Summary and Thesis Outline 20
1.4.1.1 The Problem 20
1.4.1.2 The Assumptions 20
1.4.1.3 The Claims 20
1.4.1.4 The Results 21
1.4.1.5 Thesis Outline 22
Chapter 2: The Architecture 24
2.1 Controlling Primitive Activities 25
2.1.1 Describing Transfer Functions 26
2.1.2 Describing Complex Transfer Functions 29
2.1.2.1 Conditional Expressions 30
2.1.2.2 Conditional Activation 31
2.1.2.3 Modularization 32
2.1.3 Channels 33
2.1.3.1 Minimum and Maximum Channels 34

2.1.3.2 Averaging Channels..................................35
2.1.3.3 Prioritized Channels..................................36
2.1.3.4 Other Types of Channels..............................38
2.1.4 State Variables..39
2.1.5 The Language..42
2.2 The Sequencing Layer..43
2.2.1 The Problem..44
2.2.2 Cognizant Failure..45
2.2.3 Interfacing to Primitive Activities...............................46
2.2.4 Tasks Schemas, Tasks, and the Task Bin......................48
2.2.5 Task Sequencing..49
2.2.6 The Wesson Oil Problem.......................................50
2.2.7 Real Time and Caching Computations.........................52
2.3 The Deliberative Layer..53
2.4 ATLANTIS: A Summary..57
2.4.1 The Control Layer..57
2.4.2 The Sequencing Layer..58
2.4.3 The Deliberative Layer...59
2.4.4 Who's in Charge?...59
Chapter 3: A Language For Action..62
3.1 Data Types..62
3.2 Modules...64
3.3 Channels..66
3.3.1 Channel Examples..67
3.3.2 The implementation of channels...............................68
3.4 State Variables..69
3.5 Other Features..70
3.5.1 Syntactic Abstraction...70
3.5.2 Timers and Assertions..71
3.5.3 Type abstraction..72
3.6 The Compiler..73
3.7 Summary...74
Chapter 4: Discussion and Evaluation..76
4.1 Robustness..76

4.2 Limited Computational Resources 79
4.3 Unpredictable Environments 81
4.4 Unreliable Sensors and Actuators 82
4.5 Related Work 85
4.5.1 Subsumption 85
4.5.1.1 Brooks' Architecture 86
4.5.1.2 Connell's Architecture 87
4.5.1.3 Critique 88
4.5.2 RAPs 89
4.5.3 TCA 91
4.5.4 AURA 92
4.5.5 Situated Automata 93
4.5.5 Payton and Rosenblatt's Architectures 94
4.5.6 Pengi and Sonja 95
4.5.7 PRS 96
4.5.8 Other Architectures 96
4.6 Summary 98
Chapter 5: Experiments 101
5.1 Tooth 102
5.1.1 The Robot 102
5.1.2 Experimental Setup 104
5.1.3 The Program 105
5.1.4 Results 109
5.2 Toto 110
5.2.1 The Robot 111
5.2.2 Safety Modules 112
5.2.3 Following Walls 114
5.2.3.1 Deriving the Control Law 115
5.2.3.2 Discussion and Analysis 120
5.2.3.3 Providing Cognizant Failure 125
5.2.4 Going Through Doors 127
5.2.5 The Sequencing Layer 128
5.2.6 Experiment 1 129
5.2.7 Experiment 2 132

5.2.8 Experiment 3 136
5.3 Robbie 138
5.3.1 Navigation 139
5.3.2 Pan-tilt Control 142
5.3.3 Path Planning 143
5.3.4 Vision Processing 143
5.3.5 The Sequencing Layer and the Deliberative Layer 143
5.3.6 The Experiment 144
5.4 Simulator Experiments 148
5.4.1 Experiment 1: Replication of Arroyo Results 148
5.4.2 Experiment 2: Complex Navigation 149
5.4.3 Experiment 3: Rocks and Martians 152
5.4.3.1 Martian Physics 152
5.4.3.2 Algorithm Changes 153
5.4.3.3 Results 155
5.4.4 Experiment 4: Corridors 161
5.5 Summary 164
Chapter 6: Conclusions 166
6.1 Summary 166
6.1.1 The Problem 166
6.1.2 Activities 167
6.1.3 ALFA 168
6.1.4 ATLANTIS 169
6.1.5 Experiments 170
6.2 Discussion 170
6.3 Future Work 173
6.4 Final Words 176
Bibliography 178
Appendix: Annotated Run 184
Vita 241

Chapter 1: Introduction

Driving a car is a difficult task which takes people a considerable amount of time to master. The car's velocity and heading must be constantly controlled to avoid collisions while at the same time leading towards a destination. The driver must respond quickly to unexpected contingencies like animals running across the road, while at the same time planning ahead to decide, for example, whether to get gas at this exit or wait for the next one.

This thesis is about getting a robot to drive. We will not literally seat an anthropomorphic robot behind the wheel; instead, we will deal with autonomous mobile robots which move about on their own. An autonomous mobile robot navigating about in the world might correspond better to a person walking than driving. However, driving seems to lend itself more easily to introspection, and so I will retain that analogy. The mobile robot does not correspond to the driver alone, but rather to the car and driver as an integrated system. The driver can be construed as corresponding to the robot's computer and sensory system, the car to the robot's actuators.

Driving is hard for three fundamental reasons. First, the amount of time available to decide what to do is limited. If a deer runs in front of my car I have only a few seconds to react. Even processes which take a long time such as planning a route from a map have time limits; it does me no good to plan an optimal route to the beach if the process of planning uses up my entire vacation. Second, the world is unpredictable[1] to a large extent: animals run in front of cars, tires go flat, roads are closed for repairs. This usually makes it impossible to plan a complete and reliable course of action in advance. Third, sensors and actuators are not perfect. Sensors are noisy and sometimes just flat out wrong. Actuators are imprecise. Error accumulation and discontinuities in the world can turn mere imprecision into outright failure.

These problems are fundamental problems because they cannot ever be engineered away. No matter how powerful a computer we build, a finite amount of time will allow only a finite amount of computation. No matter how good a sensor we may build there is

---

[1]Throughout the thesis, the word *unpredictable* will be used to mean *partially* unpredictable. The world is never completely unpredictable, but continuously qualifying the word becomes tiresome quickly.

always information that it cannot deliver because the relevant situation is hidden behind a wall or across town. No matter how many theories of commonsense physics we may postulate, some aspects of the world simply cannot be predicted even with perfect knowledge and unlimited computing power. Limited computation and sensing makes the problem all that much worse: there is simply no way for me to predict now whether or not I will have to stop for the traffic light at the corner when I leave work today.

This thesis addresses the problem of how to design a control mechanism for an autonomous mobile robot that will allow it to reliably move about the world in the face of these three problems, namely:

- Limited computation time
- A largely unpredictable environment
- Imperfect sensors and actuators.

The behavior of the robot should display three characteristics. It should be *reactive*, that is, it should be able to respond quickly to unexpected contingencies, both failures and unexpected opportunities. Second, it should be *task-directed* and *taskable*, that is, the robot should choose actions which tend to help it achieve its goals, and those goals should be easy to change. Third, the robot's behavior should be *robust* and reliable. We should be able to let the robot go off on its own and be fairly confident that it will achieve its goals, or, at the very least, that it won't get into trouble.

In order to describe in more detail the sorts of behavior that the system should exhibit I will begin with an extended example to which I will return often. The example is a story of an actual driving experience which occurred during the course of this research. It isn't very scientific. It isn't even a very good story, but it will help to illustrate many of the points of the thesis in a familiar way.

1.1 Driving to Hoboken

In the summer of 1990 I drove from Somerville, Massachusetts, a suburb of Boston, to Hoboken, New Jersey. I set out at 8:00 in the morning armed with a map of Southern New England and the following set of instructions given to me by the friend I was visiting:

> Go down I-95 for four or five hours. Follow the signs for the Bruckner expressway. Take the bridge to Manhattan. (It's either the Triborough or the Washington bridge; I forget which.) As soon as you get over the bridge take an immediate right and get on to Roosevelt drive. Get off Roosevelt drive at 42nd street. All the streets in New York run east-west while all the avenues run north-south. Ignore Broadway; it's weird. Take any street across town and follow the signs to the Lincoln tunnel. Go through the tunnel and get in the extreme right lane. Take the exit where the sign directs you to Hoboken. When the road straightens out you should be on Willow avenue and there should be a cliff to your right. Turn at the first legal left turn after 13th street (probably 12th street). Take your first left onto Clinton. I'll meet you at the Billabong bar which is at the corner of thirteenth and Clinton.

Getting to Massachusetts Avenue

In addition to getting to Hoboken, I also wanted to visit Yale University in New Haven, Connecticut. Instead of going down I-95 I decided instead to head West on the Massachusetts Turnpike until I got to I-91, then go South on I-91 to New Haven which is located just north of where I-91 and I-95 meet. Unfortunately, my map was not detailed enough to show how to get to the Turnpike from my house. However, it did show that the Turnpike intersected with Massachusetts Avenue, a major artery running from Boston through Cambridge and Somerville. I knew roughly where Massachusetts Avenue was, so I boldly got in my rented car and drove off. Although it was less than half a mile away from my house near Davis Square, it took me nearly fifteen minutes to get to Massachusetts Avenue because I was unfamiliar with the area, and several times encountered one-way streets going the wrong way.[2]

Getting to New Haven

Once on Mass. Ave. I followed it through Cambridge, across the Harvard Bridge, and on to the Mass. Pike. The trip to New Haven passed uneventfully. I found Yale, picked up some papers, and resumed the trip.

[2]Traffic would flow much more smoothly if only they would fix those backwards one-way streets!

The First Traffic Jam

A few miles past New Haven road construction narrowed I-95 down to one lane. There was about a one-mile backup, and a lot of stop-and-go driving and dodging. I briefly contemplated getting off the highway and taking surface streets to get around the construction, but decided against it.

Stopping for Gas

I got to the Bronx, which is the northernmost part of New York City, about six hours after I had left home. My gas tank was still about 1/3 full (I had rented a Volkswagen) but I decided to stop for gas anyway because trying to find gas inside New York city was too scary to contemplate. This also gave me an opportunity to buy a map which revealed that the bridge between the Bronx and Manhattan was the Triborough and not the Washington. I proceeded down I-95 (after missing the on-ramp once and having to make a U-turn), crossed the Triborough bridge, and took my first right onto a street marked "F.D.R. Drive".

F.D.R. Drive; The Second Traffic Jam

F.D.R. Drive turned out to be more like a freeway than a city street. It had limited access ramps, and the exits were all onto numbered streets which progressed monotonically until I got to the 42nd street exit. I took 42nd street West through Manhattan until I again encountered road construction and associated traffic. This time I decided to try to get around the problem. I turned left and drove South for several blocks until I got to 32nd street which didn't look quite so congested. I turned right and drove West on 32nd street until I saw a sign for the Lincoln Tunnel. The signs led me into one of the many mazy entrances to the tunnel. It was the height of the rush hour now, and it took the better part of an hour to get through the tunnel.

Missing the Exit

On the other side of the tunnel I encountered a major problem. There was a sign which said "Hoboken next left", or so I believed at the time. This conflicted with my instructions which said to take the next right. I decided that the sign probably knew what it

was talking about and that I might have made an error transcribing the directions. Several miles later, when I had seen no exit to Hoboken on either side of the street, I decided that something had gone wrong.

Dead Reckoning in New Jersey

I took the next exit and attempted to drive in the general direction where I knew Hoboken to be: behind me and to my left. I ended up on a road which travelled roughly southwest, parallel to the Hudson River. This road had a concrete divider which prevented me from turning left except at major intersections which appeared only every few miles. Furthermore, the intersections had a feature known as "jug handles" which are lanes that peel off from the right side of the road, turn 90 degrees to the left and then cross the original road as if at an intersection. These are used to turn left, the upshot being that in order to turn left you have to be on the right side of the street. By the time I figured this out I was hopelessly lost.

Asking Directions

Eventually I managed to turn left. I was now travelling in what I presumed to be the right direction, but I had no idea where I was. I drove around for a while hoping to find a sign to direct me, but to no avail. Ultimately I stopped at a service station to ask directions. By pure luck it turned out I was just a few blocks south of my final destination. The station attendant directed me to Clinton Street. It took me about four or five passes back and forth around the Billabong pub before I was able to sort out the one-way streets and find a parking space.

1.2 Issues in Real-World Navigation

This thesis addresses the problem of how to control a mobile robot so that it could conceivably drive to Hoboken. Of course, the robots used in this thesis did not drive to Hoboken. One drove around Jim Firby's office picking up small objects. Another drove from Ian Horswill's desk to Anita Flynn's office on the other side of the seventh floor of the Massachusetts Institute of Technology Artificial Intelligence Laboratory. A third drove through the Arroyo Seco next to the Jet Propulsion Lab. A simulated version of that third

robot chased martians and delivered rock samples in a virtual world. However, there are many similarities between these tasks and driving to Hoboken. In particular, all of them involved surmounting the three fundamental problems discussed at the start of the thesis:

- Limited computation time
- Sensor and actuator errors and noise
- Unpredictable aspects of the environment

Examining the process that eventually got me to Hoboken can perhaps shed some light on the sorts of processes that are needed get a machine to move around in the real world, whether it be across the building or across the country. The following sections highlight the main themes which the thesis will address.

1.2.1 Local Sensor Data vs. Internal State

Navigation, whether it be in a hallway or on a highway, involves controlling a physical system in response to sensory input and internally stored state information. The proportion to which sensory input and internal state information are used varies, but in humans performance seems to drop as one approaches either extreme. I can't drive with my eyes closed, but even with my eyes open I had trouble finding my way in New Jersey where I knew almost nothing about the environment beyond what I could directly perceive.

Sensor data is necessary for a number of reasons. First, the world is a dynamic and unpredictable place. Cars on streets and people in hallways move in capricious ways, and in order to navigate without collisions it is necessary to monitor the surroundings constantly.

Even if the world were perfectly static, sensors would still be necessary. Because of mechanical uncertainty, it is impossible to build a system which navigates reliably without some sort of feedback. Wheels slip on carpets. Tiny errors in heading can lead to large errors in position. Therefore sensory information is necessary, if for nothing else, to provide feedback for the low-level control mechanisms driving the robot.

On the other hand, internal state is often needed in addition to local sensor data for robust goal-directed behavior. Sometimes the information necessary to decide what to do

to achieve a goal is simply not available to local sensors. For most of my trip there was no information available to my local sensors to indicate that the way I was headed would lead me to Hoboken; I had to rely on instructions and maps, i.e. plans and world models.

This point is somewhat controversial. Connell has argued that state information is not necessary for goal-directed behavior. As proof he exhibits an actual robot which moves about in the real world and collects soda cans [Connell89]. In fact, Connell claims that state information is often detrimental to a robot because it tends to propagate sensor error over extended periods of time. While it is true that state information can cause propagation of sensor errors, I will argue that the answer is not to abandon internal state, but rather to manage it more carefully. This is a key issue, and I will return to it often.

1.2.2 Dealing with Sensor Noise

While sensory information is necessary, it is often noisy or just plain wrong. Though I did not actually go back and check, I strongly suspect that if I were to go back to the sign at the exit of the Lincoln Tunnel and examine it more carefully I would find that it does indeed say "Hoboken - next right" rather than "Next left" as I perceived at the time. Robots are particularly vulnerable to noise because their sensors are typically of much lower quality than those humans have at their disposal[3]. Due to erroneous sensor data or a changing world, the robot's internal state information may be wrong as well. An effective navigation scheme must be robust in the face of sensor noise, and must be able to detect erroneous internal state information and update it based on newly acquired data.

1.2.3 Real-Time Response

All computations in navigation are time-limited, but some are more limited than others. When the New York cabbie slams on his brakes in front of you there are typically only a few seconds (sometimes less than a second) to respond before colliding. An autonomous robot must operate at the pace of the world. A robot that does the right thing is useless if it does it too late. Elevator doors and oncoming trucks wait for no theorem prover.

---

[3]We can build cameras which are optically superior to the human eye, but for signal processing capabilities we can't touch the visual cortex.

On the other hand, sometimes there are decisions that can and do take longer to make. The decision to go down I-91 rather than I-95 required many minutes of poring over a map. The decision not to attempt to skirt the traffic jam on I-95 was not made instantaneously, but required some time-consuming consideration of the likely outcomes of various courses of action. However, even this decision had a deadline; it would not be very useful to decide to try to avoid the traffic jam once I was already past it.

1.2.4 Dealing with Failure

Humans make mistakes. It is easy to jump to the conclusion that the mistakes humans make are the results of some kind of shortcoming of the human brain. The wetware, being somewhat *ad hoc* in its design, is an imperfect implementation of a perfect algorithm, or so goes the argument. The same algorithm implemented on our carefully designed and error-free silicon brains should be able to do everything that humans do and without the annoying blunders. This line of thought has been the unstated assumption in a large portion of work in artificial intelligence (AI). Every paper that talks about anything "optimal" or "sound" or "complete" or "proven" makes this assumption.

Some researchers have argued that this assumption is invalid. They point out that the amount of computation required to produce a provably correct solution to even a trivial planning problem using perfect input data is exponential at best and unbounded at worst. For example, Chapman has shown that planning is undecidable if one admits operators with conditional effects [Chapman87]. Therefore, any robot operating under time constraints (e.g. any robot navigating in the real world) will occasionally make mistakes.

Humans almost always manage to get where they are going not because they never err, but because when they do they are able to tell that something has gone awry and take steps to recover. Expecting our robots will always do the right thing is expecting too much. Rather, we should design them to *fail cognizantly*, that is, to detect and recover from the failures which will inevitably happen. Sometimes our robots will even get hopelessly lost and have to stop to ask for help (especially in New Jersey). Realistic navigation cannot be provably correct.

1.2.5 The Role of Plans

The instructions that my friend gave me over the phone are a fairly typical example of the sort of instructions which humans often employ in getting someplace unfamiliar. Much of Artificial Intelligence is based on the assumption that in our brain is a computational structure corresponding to these instructions called a *plan*. A plan, according to the theory, can be viewed as a computer program, a step-by-step algorithm which may be mechanically executed by an automaton. As a result, much of the work on intelligent agents has focused exclusively on generating plans, since, if one accepts the plan-as-program assumption, executing a plan is very uninteresting. One simply follows the directions; any moron could do it.

The plan-as-program theory has come under attack in recent years because it does not provide an adequate account of many of the things people use plans for [Agre90]. Consider the instructions on my trip. The first "step" in the instructions was to drive down I-95 for four or five hours. There is an implied precursor step of first getting to I-95, the details of which are completely unspecified. I happened to get to I-95 by going through New Haven, by which point there were only about two or three hours to travel before going on to the next step. If I had followed the instructions to the letter at that point I would have ended up in Delaware. Then there are sentences like, "Ignore Broadway; it's weird." How and when does one execute that step in the program?

An alternative proposed by Agre and Chapman is to consider plans not as programs to be executed, but as more general resources to be used as the situation demands. Thus, plans are just a general method of communicating information which may be useful in achieving a goal. The sentence, "Ignore Broadway - it's weird," becomes not a step to be executed, but a piece of knowledge to be invoked if I encounter Broadway and need to decide what to do. Knowing that I should ignore it might not tell me unequivocally what I should do, but it might help me make the decision. Plans are used to guide actions, but not to control them. This thesis presents an implemented system which uses plans in exactly this way.

1.3 Actions vs. Activities

It is common in the AI literature to decompose actions hierarchically. Higher-level actions consist of sequences of lower-level actions. The hierarchy bottoms out in *primitive* or atomic actions which can be directly executed. The concept of low-level and high-level is very useful, but the notion of action is problematic.

1.3.1 What are Actions?

The word *action* as used in the AI literature can denote two different concepts: there is the physical action, and then there is the internal computational structure representing the physical action. The latter is sometimes called an *operator*, a term which I will adopt to avoid ambiguity[4].

According to the classical paradigm, an action is produced by *executing* an operator. In order to distinguish between this technical notion of action as the physical activity produced by the execution of an operator, and the idea of physical activity in general, I will capitalize the former. Thus, executing an operator produces an Action, but an action might be produced in other ways.

There is a very close correspondence between Actions and operators, which is one reason the terms are sometimes used interchangeably. The process of execution is atomic, that is, it cannot be decomposed. Execution takes place inside a black box, and the only way to use the box is to send it an operator, close your eyes, and wait until it's done. An operator may fail, that is, it may not produce the Action (nor the action) that it was intended to produce. But there is no way to stop or change the execution of an operator in the middle.

The formalisms of operators and Actions came about not because they are a particularly good way to describe action, but rather because the plans-as-programs

---

[4]This terminology is not universal. For example, [Nilsson80] uses the term operator to refer to a schema with uninstantiated variables, while an action is used to refer to an operator whose variables have been bound. There is no word in Nilsson's ontology for the physical action produced by executing an instantiated operator.

assumption mandates it. Planning is (usually) defined as computing a sequence of operators. Operators are the stuff that plans-as-programs are made of. Of course, there are exceptions (e.g., [Miller85], [Hogge88], and [Dean88]), but the majority of work on autonomous agents in AI has been based on operators.

The problem with Actions and operators is that they make it very difficult to talk about two actions which occur simultaneously. One could envision an execution module that allowed more than one operator to be executed at once, but the atomic nature of execution requires that the temporal extent of the execution be the same for both Actions. This makes it impossible to, say, turn left and slow down at the same time unless turning and slowing down take the same amount of time. (Actually, slowing down is quite problematic to model as an operator even by itself.)

There are various ways to patch the operator/Action model to get around this problem. One way is to relax the strict one-to-one correspondence between operators and Actions. Thus, an operator may be *invoked* (i.e. its execution begun) at any time, but it is not necessary to wait until the resulting Action is complete before invoking another operator. This is a step in the right direction, but it leads to a number of problems. For example, how do we guarantee that two Actions which occur at the same time will not interfere with one another? What happens if an Action fails? Can Actions be interrupted, and if so, how?

One proposed solution to this problem is to allow Actions to cause changes which occur after the Action itself has finished executing. For example, the Forbin planner [Dean88] allowed Actions like, "Start the widget-making machine." This Action takes only a short time to execute, but the widget-making machine keeps on making the widget for a long time. In the interim, the robot can go off and do other things, including stopping the widget-making in the middle. Much of the work on temporal logics (e.g. [Allen83]) deals with this issue as well.

There is a minor problem, however. Suppose that turning on the widget-making machine involves manipulating a large number of controls, and thus requires a minute or two to complete. Even though the process of making the widget is interruptible, turning on the widget-making machine is not (because it is an Action). Thus, if a fire started while the robot was turning on the widget-making machine, it would have to wait until turning on the widget maker was done before it could deal with the fire.

Of course, it is easy to fix the representation to handle this problem: simply define a new operator which, rather than monolithically starting the widget-making machine, starts up the *process* of starting the widget-making machine. This way, a complimentary operator can be defined which aborts the widget starting in the middle (to deal with a fire, say) in the same way that the widget-making could be stopped by executing the "Stop the widget machine" operator.

Of course, this problem is not unique to starting the widget machine. Every Action with an appreciable temporal extent will have this problem, and we cannot know *a priori* how long a period of time might be "appreciable". Thus, we arrive at the following conclusion: no monolithic operator should ever produce an action with a significant temporal extent. Because most changes in the world take time, it follows that operators cannot change the world themselves; they can only initiate processes which cause change. Turning on the widget machine is an example of initiating a process which causes a change. Starting the process of turning on the widget machine is another.

1.3.2 Activities and Decisions

Pushing the limitation on the temporal extent of operators to such an extreme changes things enough to warrant a change of nomenclature. Rather than think in terms of Actions and operators, this thesis discusses action in terms of *activities* and *decisions*. An activity is a subset of all the actions performed by an autonomous agent in a given time period. A decision is the result of a computational process which either initiates or terminates an activity. Thus, pressing on the brake, driving to Hoboken, and singing the Star Spangled Banner are all activities. Decisions are analogous to "operators" like commence-wheel-turning whose corresponding Actions have negligible temporal extent. Activities are the actions which are initiated by decisions.

It now becomes very natural to talk about things going on at the same time. When I turn the wheel and press on the brake at the same time I make two decisions: to turn the wheel and to press on the brake. These decisions initiate two activities, that of turning the wheel and pressing on the brake. At some later time I make a decision to terminate the wheel-turning activity, and at some still later time I make a decision to terminate the braking activity.

We can also talk about composite activities. Driving to Hoboken is an activity. During the course of that activity, many other activities are initiated and terminated. However, even a complex activity like driving to Hoboken is initiated by a decision to engage in that activity.

This thesis is largely about how to organize activities. Activities can interact with each other and the environment in very complex ways. Driving to Hoboken and stepping on the brake to avoid hitting the taxi in Manhattan are related somehow. I would not have had to step on the brake if I had not been driving to Hoboken. On the other hand, my stepping on the brake was not a simply a direct consequence of driving to Hoboken either; if the taxi hadn't been there I could have driven to Hoboken without pressing on the brake at that point.

1.3.3 Levels of Activity

Hierarchical decomposition of operators and Actions has proven to be a very powerful and useful concept in the effort to get a handle on traditional planning. Intuitively it seems that we should be able to apply a similar decomposition to activities. Unfortunately, it is not quite as easy to define the concepts of "high-level" and "low-level" for activities as it is for operators. Hierarchy levels for operators are usually defined in terms of a containment hierarchy: a high-level Action consists of many lower-level Actions. To execute a high-level operator, one simply executes its component operators. The hierarchy is grounded in atomic operators which can be executed directly.

For activities the situation is not so clear. Activities can overlap in strange ways. If I start singing the Star Spangled Banner while driving to Hoboken, but do not finish until after I arrive, is the singing activity a component of the high-level driving activity or not? Furthermore, the entire notion of a high-level decision is somewhat questionable. When I make a decision to press on the brake, it means that some computational process in my brain has produced an output which causes my foot to press on the brake. This sort of interaction between the output of a computational process and a simple muscle action is fairly easy to understand. But what does it mean to make a decision to drive to Hoboken? Is this just the metaphysical aggregate of all the low-level decisions that eventually get me to Hoboken? Or is there actually a computational process which produces an output which

then, somehow, causes me to drive to Hoboken in the same way that the output of the other process causes me to press on the brake?

In order to define a useful notion of high-level and low-level activities, let us make two assumptions. First, let us assume that the computational processes that produce decisions are among the actions which can be initiated by decisions. Second, let us assume that a decision never initiates an activity which contains a copy of the computation that produced that decision. These two assumptions allow us to define a high-level decision as a decision which gives rise to an activity which includes computational processes which produce other (lower-level) decisions. A high-level activity, then, is the activity initiated by a high-level decision. High-level activities include computational processes which produce decisions. Thus, driving to Hoboken is a high-level activity because it initiated (among other things) a computational process which, at some point, produced the decision to step on the brake. This latter decision is a low-level one, since the activity it initiated included very limited computational processes. (It is not clear exactly where the hierarchy bottoms out in humans, but it isn't very important to the issues addressed here. This thesis is about engineering, not psychology.)

There are a number of characteristics by which high-level activities are distinguished from low-level ones. These will serve as useful guidelines for the design of a robot control system.

1.3.3.1 Commitment

Some decisions initiate actions whose consequences are intended to affect the agent long after the actions are over. For example, suppose that I am on a surface street approaching a freeway on-ramp. By turning the wheel at the right time, I can get on the freeway. However, once on the freeway, I cannot get off (legally) until the next exit. The activity of turning the wheel lasts only a couple of seconds, but the decision to engage in that activity commits me to being on the freeway for several minutes or more. The decision to jump off a cliff commits one to falling for a while. By contrast, once on the freeway, pressing on the brake commits me to going slower only for a few seconds. I can quickly regain my former speed at any time.

Let us refer to the amount of time that an activity commits an agent to a course of action as the commit-time of that decision. Commit-time is a fuzzy concept, and I will

make only fuzzy statements about it. It is fuzzy because, in the extreme, all decisions have infinite commit time. When I press on the brake I commit to going slower only for a few seconds, but I commit to being a certain distance behind where I otherwise might have been for the rest of the trip[5]. Nonetheless, there is a sense in which some decisions can be "undone" more easily than others.

Most decisions with a high commit time are either themselves high-level decisions, or they are made directly in service of high-level decisions. When I press on the brake, a decision with low commit-time, it is usually for purely local considerations, e.g. there is an obstacle ahead. When I turn on to the freeway it is a result of the fact that I decided to drive to Hoboken. There is some local information that comes into play as well; I decided to turn for two reasons: because I decided to drive to Hoboken, and because I saw the on-ramp in front of me. If I had not earlier decided to drive to Hoboken I might have passed the on-ramp by. In other words, I might have taken different actions given the same local information, depending on the high-level decision that I made earlier.

1.3.3.2 Computation Time

High-level decisions seem to take longer to make than low-level ones. It takes longer to study a map and choose a route than to decide to stop for an obstacle. There are a number of reasons for this.

First, some decisions have deadlines imposed by the real world. The decision to stop for an obstacle must be made before colliding with the obstacle. Deciding on a route to Hoboken is usually not so urgent. The world seems to be structured in such a way that the tighter the deadline on a decision, the shorter the commit-time of that decision. In cases where this is not so, humans often run into trouble, and we have to engineer the environment to make such decisions easier to make. We put up signs at freeway exits, for example, because exits demand that high commit-time decisions be made quickly. Signs help us make those decisions more quickly.

---

[5]In the extreme, every decision changes the course of events forever. This has been the basis of a number of science fiction novels.

Second, the longer the commit-time of a decision, the more important it is that that decision be correct, i.e., that the course of action committed to by that decision not lead to undesirable consequences. The computational resources available to any agent are necessarily limited. Thus, an agent is more likely to do the right thing if it allocates those resources preferentially to thinking about those decisions which will govern its actions for long periods of time.

For example, one of the decisions that one must make when driving is which lane to drive in. When the choice is not dictated by local considerations (such as obstacles, or some lanes going to different places than others) it is better to just pick a lane at random rather than spend a lot of time deciding which lane is optimal. If the choice later turns out to be wrong, it is easy to switch to another lane. The commit-time of choosing a lane is low, and so it is better to expend scarce computational resources thinking about other things. (Another example is the decision of whether to grip the wheel with one hand or two.)

1.3.3.3 The Role of Internal State

The use of internal state in robot control architectures is the subject of an ongoing controversy. Some researchers have argued that internal state is unnecessary and even detrimental to the performance of an autonomous robot and should be avoided whenever possible (e.g. [Connell89], [Brooks86]). The argument given is that "the world is its own best model", that is, that directly sensed information necessarily provides a more accurate picture of the state of the world than a stored world model, and therefore a robot's behavior is more likely to be correct if it chooses its actions based on the former rather than the latter. If any state is used at all it should be "ephemeral" state which reverts back to some default value after a given period of time.

It is true that sensors often provide better information than internal state. However, there are cases where the information necessary to choose the correct action is simply not available locally. If I am at the end of my driveway and I want to go to Hoboken there is simply no information there that will tell me which way to go. I have to know where Hoboken is. When I get to my exit, there is usually only a sign that says "Exit". I have to remember that two miles back there was a sign that said "Hoboken - next right" in order to know that I should turn here now. There is no question that the use of internal state

presents certain problems. However, the answer is not to abandon internal state, but rather to manage it carefully and use it properly.

Let us begin by observing that there are at least four different kinds of internal state information. There is state information which contains cached sensory input, and there is state information which stores the outputs of decision-making processes. Each of these can be further classified into short-term and long-term state. An example of short-term sensory state is the memory of the "Hoboken - next right" sign. I remember seeing this sign until I get to the next right, then I forget about it. An example of long-term sensory state is my knowledge of the existence and general location of Massachusetts Avenue. An example of short-term decision state is the internal state that caused me to drive up to the gas pump after I pulled off I-95 in Brooklyn even though my tank was not empty. An example of (very) long-term decision state is the internal state that causes me to write this thesis hour after endless hour.

Obviously, high-level decisions tend to produce long-term state of the decision-cache sort. Low-level decisions produce short-term state. What is less obvious is that high-level decisions also tend to make use of long-term sensory state more than low-level decisions. Low-level decisions can use long-term state implicitly in cases where those decisions are made in service of high-level decisions, but they rarely use long-term state directly. The reason for this is that the amount of long-term sensory-cache state information that an agent may have at its disposal can be extremely large and it can take a significant amount of time to access and process it. Since low-level decisions are the results of fast computational processes, they simply don't have time to use this information.

1.3.3.4 Levels of Abstraction

An abstraction is simply a description of something. A level of abstraction is a measure of the precision of that description. A high-level abstraction is less precise than a low-level abstraction. For example, a topological description is a higher-level abstraction than a geometric description because the geometric description is more precise. "Driving to Hoboken" is a higher-level abstraction than "Driving down I-95 at 55 miles per hour".

As one describes the world at higher levels of abstraction it tends to appear more static because the dynamics of the situation get abstracted away. At a low level of abstraction the world is in constant flux. People, cars, and small objects tend to move around on a regular

basis. At a higher level of abstraction the world is more nearly static. Buildings can be built, torn down, or modified, but this tends to happen more slowly and less frequently than changes at lower levels of abstraction. At very high levels of abstraction things change extremely slowly. It is fairly certain that Hoboken will still exist five years from now and, moreover, that it will still be more or less in the same place that it is today.

The reason abstractions are useful is that the computational processes which produce high-level decisions are slow. Therefore, they cannot deal effectively with dynamic situations, especially if the dynamics are unpredictable. If the world were constantly changing in all of its aspects at all levels of abstraction then no high-level decisions would be possible because, by the time the computation was complete, the world might have changed in such a way that the decision would be wrong. It would do no good to choose a particular route to Hoboken if there were a good chance that Hoboken would move someplace else before arriving.

High-level decisions are only effective if they are made at a high level of abstraction. They are effective because the world is mostly static at a high level of abstraction.

1.4 An Architecture for Autonomous Navigation

The preceding sections have described some intuitions and hypotheses about the nature of activities. This thesis is mainly about how to translate these intuitions into a computational mechanism for controlling autonomous mobile robots. This section gives an overview of how this will be done.

To summarize briefly, the central hypothesis of this thesis is that activity in general and navigation in particular consist of a hierarchy of decisions and activities. High-level decisions are made slowly, with reliance on internal state, at high levels of abstraction, and initiate activities which last a long time. The activities initiated by these high-level decisions include computational processes which run fast, rely mostly on local sensor information, operate at low levels of abstraction, and initiate activities which don't last very long. The hierarchy bottoms out in decisions which initiate activities which include no computational processes, only physical processes.

This thesis attacks the problem bottom-up. It begins by describing the computational structure required to control primitive activities, or activities which contain no decision-

making computations. Primitive activities have low commit time, use very little internal state, rely on fast computations, and operate at a low level of abstraction. Primitive activities are controlled by computations which use sensor information as input and produce actuator commands as output. This sort of computation requires a different computational infrastructure and abstraction mechanism to support it than that provided by traditional programming languages. A new programming language called ALFA (A Language For Action) is developed to meet these requirements.

ALFA is then used as the bottom layer of a three-level architecture for robot control called ATLANTIS[6]. The first layer controls primitive activities. The second layer is responsible for controlling sequences of activities. This layer is essentially an operating system. Time-consuming computational activities (such as planning) run asynchronously in the third layer. Most of ATLANTIS is constructed out of existing technology.

The main result of the thesis is that different sorts of computational structure are required for supporting different levels of activities. The sorts of structures which are useful for controlling primitive activities are not very good for supporting high-level activities. Thus, the architecture developed in this thesis is *heterogeneous*, that is, it consists of a number of different components each of which is structured differently from the others. ALFA is designed specifically to be a component in such an architecture.

The performance of the system is experimentally verified on both real and simulated robots. The performance of ALFA alone is demonstrated on a small indoor robot which performs an object-collecting task. The performance of ALFA in conjunction with a rudimentary higher-level system is demonstrated on an indoor robot which performs a complex navigation task.

The performance of a more sophisticated version of the higher-level system is demonstrated on the JPL planetary rover testbed, as well as on a real-time simulation of the testbed. While the internal structure of the low-level control was different on this robot, the interface was made to look as if it were an ALFA program. This robot and its simulated counterpart performed complex tasks requiring the use of deliberative computations as well as reactive control.

---

[6]A Three Layer Architecture for Navigating Through Intricate Situations

1.4.1 Summary and Thesis Outline

In this section I review as succinctly as possible the major points of the thesis.

1.4.1.1 The Problem

This thesis addresses the problem of how to control autonomous mobile robots in unaltered real-world environments in the face of sensor noise, unpredictability, and limited computation in a way which is reactive, robust, and task-directed.

1.4.1.2 The Assumptions

A viable control mechanism must be based fundamentally on a continuous-action model. Computations and physical actions collectively form *activities* which are initiated and terminated by *decisions*. Decisions are the results of computational processes which are themselves activities.

Activities and decisions can be arranged in a hierarchy. Four characteristics distinguish among various levels. Higher-level decisions 1) require more computational resources, 2) are made at a higher level of abstraction, 3) rely more on internal state information, and 4) commit the robot to a course of action for a longer period of time than lower-level decisions.

1.4.1.3 The Claims

1. A successful architecture for controlling autonomous mobile robots should be *heterogeneous* and *asynchronous*, that is, it should have components which are structured differently from one another, and which operate in parallel. The architecture should include mechanisms for both deliberative computations and reactive control.

2. ALFA is a suitable mechanism for implementing the reactive portion of such an architecture because of its dataflow semantics, uniform communications model, and internal structure. The remaining portions of the architecture can largely be implemented using existing technology.

3. A reliable system for controlling autonomous mobile robots in unaltered real-world environments can be constructed by engineering activities bottom-up to *fail cognizantly*, that is, to detect failures when they occur.

4. Internal state information is necessary in many situations to produce robust goal-directed behavior. State information should exist at a high level of abstraction. Existing technology for manipulating such information can be successfully integrated into the overall system.

5. The sensory information and control mechanisms required to keep the robot out of danger are qualitatively different from those required to produce goal-directed behavior.

1.4.1.4 The Results

A heterogeneous, asynchronous architecture for controlling autonomous mobile robots was developed. This architecture includes both deliberative and reactive components. The architecture is structured in a way which allows existing AI technology to be incorporated into the deliberative component.

The computational infrastructure for the reactive component was studied in detail. A programming language was designed to help deal with the complexity of controlling primitive activities. A compiler for this language was implemented.

Partial implementations of the architecture were used to control three very different real-world robots in real-world environments: a small, indoor robot developed at JPL, another indoor robot developed at MIT, and the JPL planetary rover testbed. These robots exhibited robust, goal-directed behavior using noisy sensors and actuators, limited computation, and with only partial or non-existent prior knowledge of the environment.

In addition, the architecture was extensively tested on a real-time simulator which simulates the planetary rover. The results of the experiment performed on the actual robot were reproduced on the simulator to provide "ground truth" evidence that results produced by the simulator bear some resemblance to reality. The system was then used to perform a variety of tasks, often under conditions much more severe than those which exist on the actual robot.

The robots in these experiments display many of the features in the Hoboken story. For example, the simulator robot follows instructions at a very high level of abstraction

such as, "Collect 3 green rocks and bring them to landing site." It also stops for fuel when it might not have enough to complete its next task. The MIT robot was able to follow a set of instructions on the order of, "Go to a place near the office door. Go through the door. Go northwest until you find a wall. Follow the wall until you come to the second office on the left. If you lose sight of the wall, stop and ask for help." The planetary rover testbed was able to replan its route when it encountered a "traffic jam" in the form of an experimenter blocking the way to one of its goals. All of these robots display robust, reliable behavior.

1.4.1.5 Thesis Outline

This thesis is divided into six Chapters.

Chapter 1 is the Chapter that you are currently reading. It presents an informal theory of action upon which the rest of the thesis is based.

Chapter 2 describes ATLANTIS, a heterogeneous, asynchronous control architecture for a mobile robot which includes both reactive and deliberative components. This chapter concentrates on the features of the architecture which affect the reactive component.

Chapter 3 describes ALFA, a programming language for writing programs to control primitive activities. Unlike traditional programming languages which perform sequential computations, ALFA programs compute highly nonlinear transfer functions from sensor input to actuator output. This Chapter describes the language constructs and the compiler.

Chapter 4 assesses how well the architecture addresses the issues outlined in Chapter 1, namely how to achieve robust control of real robots in the face of limited computation, sensor noise, and an unpredictable environment. This Chapter also compares the architecture with previous work in the field.

Chapter 5 describes a number of experiments using the architecture to control mobile robots performing complex navigation tasks both in the real world and in simulation. Portions of the architecture have been implemented on three real robots, as well as a sophisticated real-time simulator.

Chapter 6 summarizes and presents conclusions. There are four main results:

- Reaction and planning can be successfully integrated into an overall architecture for controlling mobile robots. The architecture should be heterogeneous and asynchronous. This allows existing AI technology to be incorporated into the deliberative component without adversely affecting the response time of the overall system.
- A language for programming reactive control mechanisms which are designed to interface to higher-level systems was designed and implemented.
- A design methodology for controlling mobile robots was developed. The methodology is based on bottom-up hierarchical decomposition and cognizant failure.
- The system has been experimentally demonstrated on three real robots and a simulator.

This Chapter also makes some comments about the lack of formal theory in the thesis and suggests some directions for future research.

All subsequent Chapters rely on material presented in Chapter 1. Chapter 3 relies a bit on Chapter 2, especially section 2.1. Beyond that, the Chapters are more or less independent. Read Chapter 2 if you are interested in the technical details of the architecture. Read Chapter 3 if you want to know the details of how primitive activities are controlled. Read Chapter 4 if you want a general idea of how the architecture does what it does and how it compares to other work in the field. Read Chapter 5 if you want to know how ALFA and ATLANTIS performed, what sorts of problems one tends to encounter on real robots, and what sorts of features a simulator should have in order to (possibly) produce useful results. Read Chapter 6 if you want a digestified version of the whole thesis and wild speculations about the future. Chapter 6 also has an explanation of why there are no proofs in this thesis, and why this is a feature rather than a drawback.

Chapter 2: The Architecture

This chapter develops the computational structure required to implement a control mechanism for an autonomous mobile robot based on the concept of decisions and activities developed in the last chapter. To review briefly, an *activity* was defined as a subset of all the physical and computational actions engaged in by an agent over a given period of time. A *decision* is the result of a computation which initiates or terminates an activity. Activities can be classified into a hierarchy: higher-level activities include computations which initiate or terminate lower-level activities. The hierarchy bottoms out in activities which include no decision-making computations, only physical processes and their associated control algorithms.

Activities at various levels of the hierarchy can be distinguished by four characteristics: commitment time, level of abstraction, reliance on internal state, and computation time. The higher one goes in the hierarchy, the higher each of these measures of an activity becomes. Driving to Hoboken is an example of a high-level activity; pressing on the brake is an example of a low-level activity.

This chapter approaches the problem bottom-up by considering first how to control primitive activities, that is, activities which contain no decision-making computations. Even by itself this is an extremely complex problem. It subsumes, among other things, the field of control theory. There are, therefore, many aspects of the problem which this thesis cannot address in detail. This chapter will put aside most of the control-theoretical issues and approach the problem instead as a software engineering problem. (The reader is referred to [Ogata87] for detailed treatments of the control theoretical issues.) The following issues are central: 1) What sort of computational infrastructure needs to be provided in order to make it easy to program primitive activities? 2) How should primitive activities be interfaced to decision-making processes? 3) How should primitive activities be structured in order to increase the probability that the system will actually work as it was intended to?

Having developed the structure for controlling primitive activities, the chapter will go on to examine how to control higher-level activities. The computational structures needed to control higher level activities turn out to be very different from those needed to control primitives. Fortunately, much existing technology can be employed for this purpose.

2.1 Controlling Primitive Activities

This section describes the first component of the ATLANTIS[1] architecture, the one that controls primitive activities. In Chapter 1 the canonical example of a primitive activity was turning a steering wheel. Turning the wheel is a good example of a primitive for a person driving down the road, and if we had an anthropomorphic robot to sit in the driver's seat, turning the wheel would be a good example for it too. However, most mobile robots do not have complex kinematic linkages between their actuators and their direction control. It is fairly straightforward to "turn the wheel" on a mobile robot because "the wheel" is usually connected directly to a motor over which the robot exercises direct control. Most of the hard problems in this case arise from the sorts of control-theoretical considerations which I am not addressing here. To make things interesting from a software-engineering point of view we need to move up a level of complexity. Therefore, for this section I will use the task of following a road (or, the indoor equivalent, following a hallway) while avoiding obstacles as my canonical example of a primitive activity.

As was argued in Chapter 1, a mechanism for controlling primitive activities should have the following characteristics:

- Fast response time
- Low level of abstraction
- Low commit-time
- Low reliance on internal state

In other words, the mechanism for controlling primitive activities should compute a (mostly) stateless, temporally continuous transfer function from the robot's sensory inputs onto its actuator outputs. "Temporally continuous" means that the mechanism must compute a value for all of the robot's actuators at every instant in time. The output of this mechanism is to be connected to real motors, and so it is not acceptable for the mechanism to ever output "undefined" or "computation pending". "Temporally continuous" does *not* mean continuous in the usual mathematical sense of continuous function. In general, the

[1]ATLANTIS: This Layered Architecture Navigates Through Its Surroundings

transfer functions required to control mobile robots in the real world can be quite complex, highly non-linear, and discontinuous. In other words, the output may change abruptly, but there will always be an output. The central issue which I will address in this section is the development of design tools which can help us manage this complexity.

2.1.1 Describing Transfer Functions

We need to be able to describe complex transfer functions from sensor inputs to actuator outputs. To do this we must be able to specify three things:

- The function to be computed
- The source of the function's inputs
- The destination of the function's output(s)

For a trivial example, let us suppose that we have a robot with a single sensor and a single actuator. The actuator controls the robot's speed and the sensor delivers the range to the nearest obstacle in front of the robot. (This robot lives in Lineland; it has no direction control.) One possible control law for controlling this robot is simply to pass the value of the range sensor directly to the actuator. This way, the robot will slow down as it approaches an obstacle and stop just as it makes contact. In other words, we want to output to the actuator the value of the sensor. I will adopt the term *drive* to denote the act of outputting a value to an actuator. Thus, we want to be able to say something like:

```
(DRIVE THE-ACTUATOR (value-of THE-SENSOR))                    (2-0)
```

Because there is nothing that can be done with a sensor besides read its value, we can drop the `value-of` notation and simply refer to the sensor by name:

```
(DRIVE THE-ACTUATOR THE-SENSOR)                               (2-1)
```

This expression looks like a command in a traditional programming language, but there is a significant difference. In a traditional programming language, expressions are *executed* or *evaluated* in a sequential fashion. You can identify discrete time intervals over which expressions and their various components are actively being processed. By contrast, we want the expression above to indicate something like a hardwired physical

connection between THE-SENSOR and THE-ACTUATOR. We can imagine that there is a wire coming out of the range sensor which carries a voltage proportional to the distance to the nearest obstacle. The expression THE-SENSOR indicates the voltage on this wire. A DRIVE command functions something like a solder joint; it connects the output of some sensor (or some computational unit) to an actuator. (See figure 2-1a.)

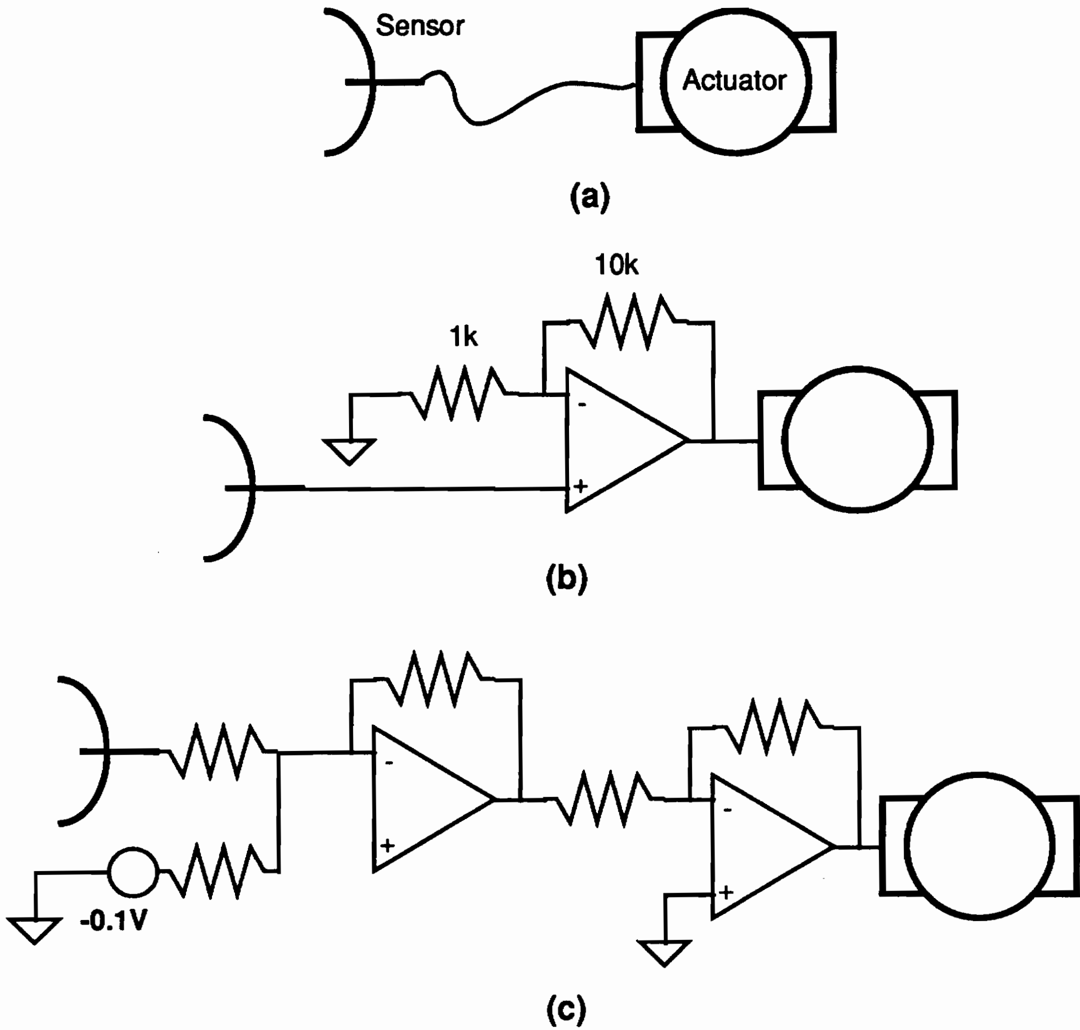


Figure 2-1: Three examples of circuit semantics.

Circuits, Programs, and the Nyquist Sampling Theorem

There are two ways to maintain the illusion that we are describing analog circuits. The first is to actually think about the control mechanism in terms of analog circuits. The second is to invoke the Nyquist sampling theorem and think about the control mechanism as a discrete approximation to an analog circuit.

The Nyquist sampling theorem states that an analog signal can be completely recovered from a discrete sampling of that signal provided that the sampling frequency is greater than 2f, where f is the frequency of the highest-frequency component in the Fourier decomposition of the signal. In other words, there is a formal equivalence between an analog circuit with a bandwidth B and a computer program computing a signal at discrete points in time provided that the computer program takes no longer than 1/2B to complete a computation cycle.

For now, I will discuss the problem in terms of analog circuits. There are two reasons for this. The first is that this will tend to keep us honest about what we should and should not be able to do in the language that we develop. Second, if we can ground the language in a semantics of analog circuits, then by virtue of the Nyquist sampling theorem we can make certain guarantees about the performance of the resulting system even when it is run as a discrete simulation on a digital computer.

A Story

Let us imagine that we have a machine that will actually wire up an analog control mechanism inside our robot. When we feed it expression 2-1, the machine reaches into the robot with a mechanical arm and solders a wire between the range sensor and the robot's drive motor. We switch on the robot and find that the robot moves too slowly for our tastes. The signal from the range sensor is not strong enough to move the motor at the speed we want. We would like to amplify the signal from the sensor before passing it to the drive motor, so we type:

```
(DRIVE THE-ACTUATOR (* 10.0 THE-SENSOR))                    (2-2)
```

We give this to our magic wiring machine and watch as it reaches into a parts bin and pulls out an operational amplifier chip and some resistors. (See figure 2-1b.) It unsolders

the previous connection and replaces it with an amplifier circuit with a gain of 10. We switch on the robot again and find that it now moves briskly down the hall. At some point it begins to approach an obstacle. It slows down, but not quite enough, and it collides gently with the obstacle. A bit of sleuthing reveals that the range sensor's output does not quite fall to zero when the robot is next to an obstacle. We need to introduce an offset into the transfer function:

```
(DRIVE THE-ACTUATOR (* 10.0 (- THE-SENSOR 0.1)))            (2-3)
```

We watch in amazement as the wiring machine builds a two-input summing amplifier with one input coming from the sensor and the other input from a -0.1V voltage source. (See figure 2-1c.) We turn on our robot for the third time and watch as it smoothly comes to a halt a few centimeters from a small child. Unfortunately, the curious child now crawls closer to the robot, causing the output of the summing amplifier to go slightly negative and making the robot move slowly backwards. The child chases the robot until it collides backwards into the wall. It seems we need some way to prevent the output of the amplifier from assuming a negative value.

2.1.2 Describing Complex Transfer Functions

The robot described above is about as simple a robot as one can imagine, and already things are getting complicated. There are two things in particular which are indicative of things to come. First, the robot behaved in ways which we did not foresee when we designed our control mechanism. The design process required some debugging. (Granted, the above story is fictional. I take it as axiomatic that debugging will occasionally be required in reality. Empirical data will bear out this assumption.) Second, the design process eventually presented us with a requirement for a nonlinear transfer function.

How do we develop our system to address these issues? We want to be able to describe complex nonlinear transfer functions in a way which will make them easy to develop and debug. We will accomplish this by introducing three mechanisms: conditional expressions, conditional activation, and modularization. Conditional expressions will allow us to write descriptions of transfer functions which are computed differently in different regions of the robot's sensor space. Conditional activation will allow us to write

transfer functions which are undefined in some regions of sensor space to be used as building blocks for constructing more complex transfer functions. Modularization will allow us to connect different transfer functions together in order to produce more complex ones.

We also want to maintain the illusion that we are describing a continuous computation. In other words, it should be possible to compile any program we write into a network of operational amplifiers and miscellaneous electronic components. There are two reasons for doing this. First, this allows us to place a strict upper bound on the time required to compute a cycle of the program when run as a discrete simulation. By virtue of the Nyquist sampling theorem, we can therefore place a strict lower bound on the bandwidth of the resulting system. This can be crucial for insuring the stability of the system. Second, if very high bandwidths are required, the program could conceivably be compiled down to analog hardware for extremely fast operation.

2.1.2.1 Conditional Expressions

In order to keep the robot in the example from going backwards we might want to write something like the following:

```
(DRIVE THE-ACTUATOR                                    (2-4)
       (IF (> THE-SENSOR 0.1)
           (* 10.0 (- THE-SENSOR 0.1))
           0.0))
```

In other words, we would like to send to the actuator a value which is 0 as long as the sensor reads less than 0.1, and is ten times the value of the sensor (less a tenth of a volt) otherwise.

This sort of conditional expression can be realized in hardware by using a comparator to check if the value of the sensor is greater than or less than 0.1 and using the output of the comparator to drive a relay (or its solid-state equivalent, an analog switch or tri-state buffer). (See figure 2-2a.)

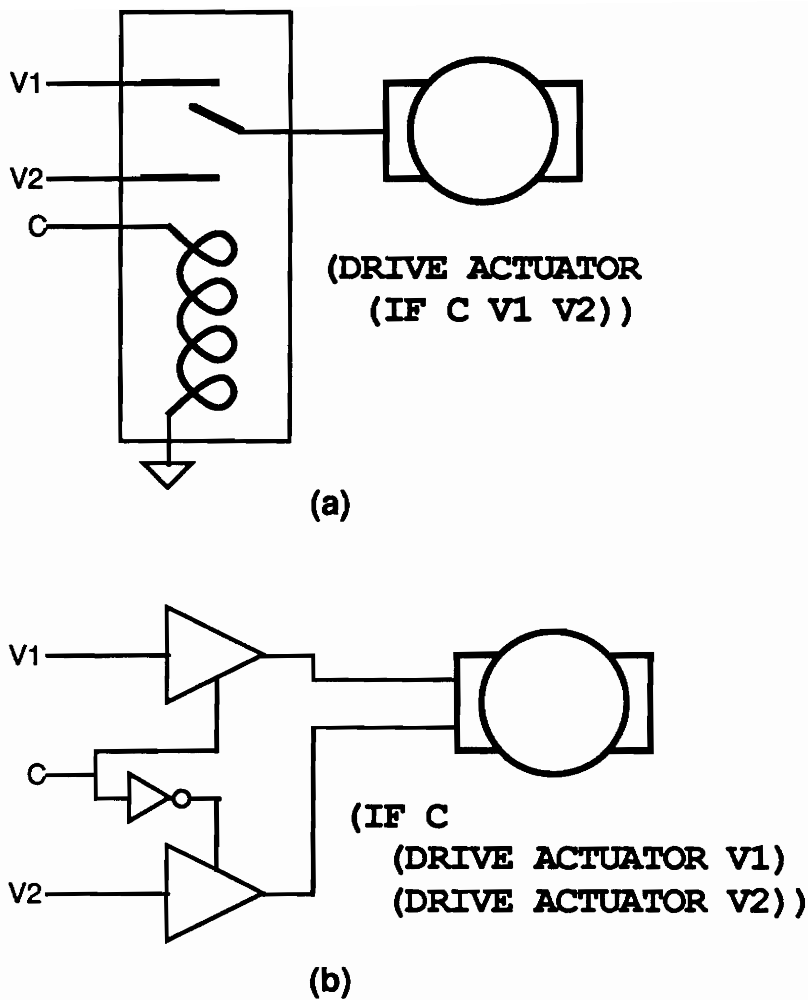


Figure 2-2: Circuit semantics for (a) conditional expressions and (b) conditional commands.

2.1.2.2 Conditional Activation

We might also have written the following:

```
(IF (> THE-SENSOR 0.1)                                        (2-5)
    (DRIVE THE-ACTUATOR (* 10.0 (- THE-SENSOR 0.1))
    (DRIVE THE-ACTUATOR 0.0))
```

In this particular case the effect is nominally the same. However, the underlying semantics are quite different. For example, suppose that we have two actuators on our robot, and we write something like the following:

```
(IF [condition]                                    (2-6)
    (DRIVE ACTUATOR-1 [expression1])
    (DRIVE ACTUATOR-2 [expression2]))
```

In the case of 2-4 we are constructing a single computational unit whose output is conditional upon the value of a boolean expression, and connecting the output of that unit to an actuator. In the case of 2-5 and 2-6 we are describing two *distinct* computational units and a *conditional connection* between their outputs and an actuator (or a pair of actuators). (See figure 2-2b.)

2.1.2.3 Modularization

Note that it is not necessary to have an `else` clause in conditional commands such as 2-5 and 2-6 for a meaningful circuit to result. This allows us to describe circuits which sometimes output nothing (i.e. go to a high-impedance state). The ability to describe such conditional connections has important consequences for the development of complex transfer functions. It allows us to decompose the development of complex transfer functions into components which need not cover the entire range of possible sensor values. This is a useful ability, but it comes at a cost. The ability to conditionally connect to an actuator means that sometimes an actuator may not have anything connected to it at all, or that it may have two different wires connected to it, each of which is attempting to drive it at a different value. We could simply be careful to construct programs that never do these things, but that seems too dangerous, and we can do better.

The solution is to introduce an intermediate device which will provide a buffer between the outputs of DRIVE commands on the one hand and actuators on the other. Let us call this device a *channel*. A channel takes an arbitrary number (possibly 0) of driving signals as inputs and produces a single output which can then be used to drive an actuator. We also need a name for the pieces of circuitry which are computing the (partial) transfer functions which get sent to the channel. In order to introduce as few preconceptions as possible about how they work I will simply call them *modules*.

The mechanism of modules and channels provides an extremely rich framework in which to develop and debug very complicated transfer functions. To see how, we must first get a little more concrete about exactly what channels do and how they do it.

2.1.3 Channels

A channel is a device which takes some arbitrary number of inputs and combines them in some way to produce a single output. (See figure 2-3.) There are many possible ways of combining inputs. A channel could average all of the incoming signals, or take the largest or smallest value, or the incoming signals could be prioritized in some way. In any case, a channel should have some default value which it outputs if no signals at all are currently being received.

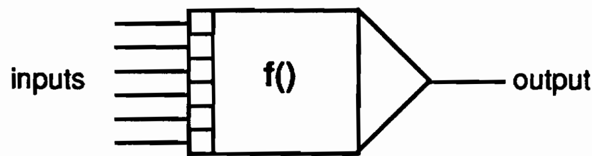


Figure 2-3: A channel.

There are countless other ways of combining an arbitrary number of inputs. One could add all the inputs together, multiply them all, take the average of successive differences, and other bizarre computations. However, the four operations mentioned above, prioritization, averaging, maximum and minimum, have exceptional utility for constructing complex transfer functions out of simpler ones. In addition, they can all be implemented in analog hardware. (See figure 2-4.)

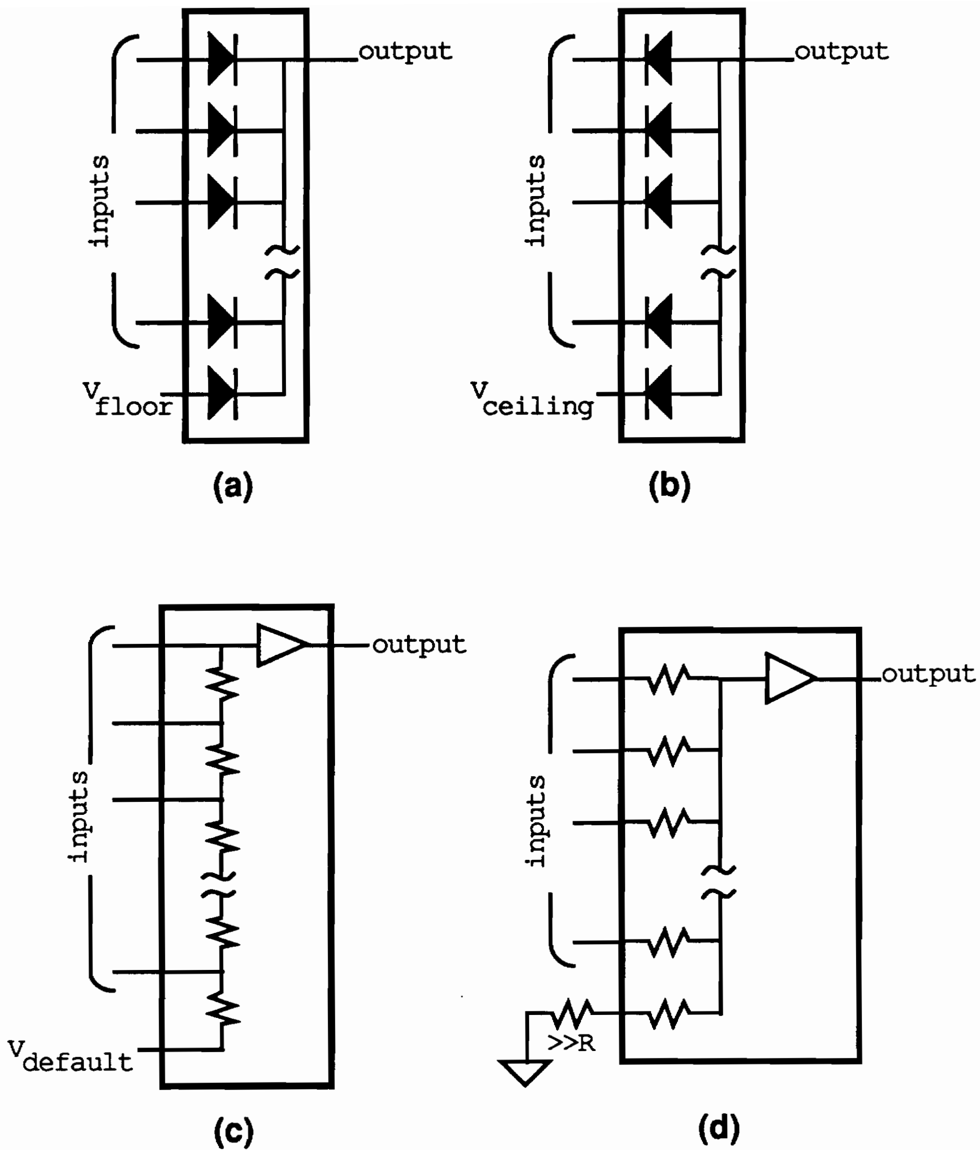


Figure 2-4: Analog hardware implementations of (a) a maximum channel, (b) a minimum channel, (c) a prioritized channel, and (d) an averaging channel.

2.1.3.1 Minimum and Maximum Channels

Let us refer to a channel which outputs the maximum of its incoming signals as a maximum channel, and conversely for a minimum channel. Maximum and minimum channels allow modules to impose constraints on the value of a transfer function without

having to specify the actual value. For example, suppose that we have a speedy robot which can go so fast that it can tip itself over during a turn. We would like to insure that the robot will not go too fast during a turn. We can arrange this by installing a minimum channel whose output controls the robot's speed. Now we arrange for any module which turns the robot to also send to this channel a value corresponding to the maximum safe speed for the turn. This will insure that the robot will not go through a turn too fast. On the other hand, it does not preclude some other module from slowing the robot down even further if the situation requires it (e.g. to stop for an obstacle).

There is one important caveat to note here. Nothing prevents the output of a minimum channel from falling below 0. Thus, if the robot can be driven in reverse, one would probably want to use two channels, one maximum and one minimum channel, so that the backwards speed could be constrained as well.

2.1.3.2 Averaging Channels

In cases where it is necessary to impose constraints on both the maximum and minimum value of an actuator one can use two channels, one maximizing and one minimizing. The two outputs need to be combined somehow to produce a value which meets both constraints. One way to do this is to simply average the two values together. In the absence of any other information averaging is the best choice because it produces a value equidistant from both constraints. In cases where the constraints contradict each other (i.e. the maximum value is lower than the minimum value) averaging produces a value which violates both constraints equally.

As a general technique for combining multiple inputs, however, averaging must be used with great care. The classic problem with averaging is the robot which, upon encountering an obstacle ahead, decides that it should go either left or right, averages the two alternatives, and ends up plowing straight ahead. To solve this problem, a type of channel is needed which selects from a group of alternatives without combining them together.

2.1.3.3 Prioritized Channels

Consider the final example from the story where the child chases the robot backwards into an obstacle. This is an example of a situation which often arises in practice where a designed transfer function does the right thing except in a particular circumstance. In other words, the function is correct except for a small range of sensor values (or, in general, in a small region of the phase space of the robot's sensor suite).

There are two ways to fix this problem. One is to change the code for the erroneous transfer function. The trouble with doing this is that, unless we are very careful, we have no guarantee that our change will not introduce a new problem. The other way to fix the problem is to construct an entirely new module which is active only within the range of sensor values where the original module was faulty. By giving this new module the ability to override the faulty one, we can construct a composite transfer function which is correct over a larger range of sensor values than either of the component functions.

There is another use for prioritized channels. Suppose that we want to ensure that our robot will always take a particular action in certain circumstances. For example, we might want to guarantee that the robot will stop when it detects an obstacle within a certain range. Simply sending a 0 to a maximizing channel controlling the speed will not work because the robot might still move backwards. Instead, we can install a module which stops the robot in such circumstances and give its output an overriding priority. If we do this then we can guarantee that the robot will stop for obstacles regardless of what other modules we may add in the future.

In order to realize these advantages we must be careful about the mechanism we use to assign priorities. There are several possibilities. For example, we might include a numerical priority in the `DRIVE` command whenever we are driving a prioritized channel. This has a number of disadvantages. First, it requires a different syntax for driving prioritized channels than for driving other kinds of channels. Ideally, we would like the syntax to be uniform so that the type of channel being driven is hidden from the module driving it. Also, it leaves open the possibility of two modules driving a single channel with the same priority, which defeats the purpose of having prioritized channels to begin with. Finally, if we do the obvious thing of having higher-numbered priorities override lower-numbered ones then we lose our ability to make performance guarantees about the overall mechanism. This is because we can never be sure that a module with a higher priority than

all the existing modules will be added and override critical signals. We could have lower-numbered priorities override higher-numbered ones and restrict the range of priorities to positive numbers, but this still leaves us with the problem of non-uniform syntax and possible unresolved conflicts.

Instead of numbered priorities we can do the following: for each prioritized channel we can list in the definition of the channel all of the modules which are allowed to drive that channel in order of priority. This allows us to maintain a uniform syntax for driving all kinds of channels, as well as retaining the ability to make certain guarantees about the performance of the system. For a new module to override existing ones the definition of the channel must be consciously changed, reducing the likelihood of introducing unexpected bugs by adding modules. Setting priorities in this way also eliminates the possibility of unresolved conflicts.

Having to cross-reference behaviors and channels in this way may seem to be a programming annoyance. I prefer to view it as a useful feature and debugging aid. Listing the driving behaviors allows certain bugs involving unintentional connections to prioritized actuators to be caught at compile time. Also, if the output of a prioritized channel is wrong then only those modules listed in the channel definition need to be examined to find the bug (assuming that the values of the other channels in the system are correct). It also allows certain assurances to be made about the performance of the system when new code is added. (See sections 2.4.1 and 4.5.1).

Another objection to this approach is that it forces the priorities to be static. We might want to design a pair of modules each of which can override the other depending on the circumstances. This objection can be countered in two ways. First, even if we were to allow dynamic prioritization we would need a static prioritization scheme as well in order to resolve conflicts when two modules drive the same channel at the same priority. Second, dynamically prioritized channels can be constructed from statically prioritized ones, so we do not preclude the possibility of dynamic prioritization by not providing it as a primitive.

A dynamically prioritized channel can be constructed from a statically prioritized one as follows. First, we introduce a second channel called an arbitrator. The arbitrator is either a minimizing or a maximizing channel depending on whether high priorities should override lower ones or vice versa. A module that wishes to drive the prioritized channel with a dynamic priority first outputs its priority to the arbitrator. It then monitors the output of the

arbitrator. Only if the output of the arbitrator is the same as the value sent there by the module does it then output a value to the prioritized channel.

However, while it is possible to build a dynamically prioritized channel out of a statically prioritized one, the utility of dynamic prioritization is questionable. Dynamic priorities are often advocated by "behavior based" approaches which use them to combine results produced by various modules which have some sort of a "certainty measure" associated with them. The intuition is that a module which is more certain of its answer should override a module which is less certain. Since the certainty measures can only be computed at run time, dynamic priorities are necessary [Rosenblatt89].

The trouble is that computing the uncertainty measures almost always requires some deliberative computations and the maintenance of a large amount of internal state (for example, to maintain previous sensor readings to compare against the current readings). Because we have explicitly rejected the use of such mechanisms for controlling low-level activities there is no basis for computing uncertainty measures and therefore no use for dynamic prioritization. The extra complexity and debugging difficulties introduced by dynamic prioritization should discourage its use in all but a few extenuating situations. All of the experiments performed in this thesis were conducted without the use of dynamic prioritization.

2.1.3.4 Other Types of Channels

Rosenblatt and Payton use a mediation technique which I will call a constraint-posting or constraint-propagation channel [Rosenblatt89]. A constraint-posting channel is a generalization of maximizing and minimizing channels in that it allows driving modules to impose constraints on the value of a channel without specifying what the actual value is. Modules can also "suggest" desirable values. For example, a module might specify that a channel may not take on any value between 2 and 5. Another module might suggest that the channel should take on a value somewhere between 4 and 6. The channel outputs a value which satisfies as many of the suggestions while violating as few constraints as possible. The example channel might output 5.5.

Constraint-posting channels come in two varieties: continuous and discrete. Continuous constraint-posting channels can assume any real-number value, and constraints can be posted in terms of arbitrary ranges of reals. Discrete constraint-posting channels can

only assume discrete values, though possibly from an infinite set, e.g. the integers. Constraint-posting channels can be further generalized to allow the constraints to be weighted by certainty values similar to dynamic priorities.

There are two problems with constraint posting channels. First, the general problem of constraint satisfaction is NP complete. To get around this, Rosenblatt restricts the constraints that can be posted to being conjunctive constraints only; no disjunctive constraints are allowed. The constraint-satisfaction problem then essentially reduces to a sorting problem, which reduces the complexity to O(n log n) time in the continuous case and O(n) time in the discrete case. However, these sorts of channels cannot be implemented in analog hardware because the number of constraints posted by a single module is not bounded. Thus, constraint-propagation channels do not meet our requirement that they must be able to perform their computations in constant time using analog hardware.

There is another reason to reject constraint-propagation channels. The utility of this sort of channel is choosing between disjoint alternative values for an actuator, that is, choosing values which may lie within one or more disjoint ranges. For example, an obstacle-avoiding module may constrain the robot to turn either right or left, but not to go straight because there is an obstacle ahead. Such disjoint alternatives often lead to disjoint regions in the robot's configuration space; these regions are separated from each other by the obstacle that caused there to be disjoint alternatives in the first place. This implies that such decisions have a high commit-time, and so choosing among them should not be part of controlling a primitive activity.

The choice of turning left or right when faced with an obstacle is a good example. If the obstacle is small it might not matter which way the robot turns to avoid it. However, if the obstacle is very large (e.g. a fence) it can make an enormous difference. To choose the correct direction in such cases often requires global knowledge and, occasionally, planning. In other words, choosing among disjoint alternatives is a high-level decision, and thus should not be performed in the control layer.

2.1.4 State Variables

Sometimes it can be useful to base an action partially on internally stored state rather than purely on the values of the vehicle's sensors. For example, suppose that we are using

a pyroelectric sensor to locate people. Pyroelectric sensors respond to the first derivative of the amount of infrared radiation which they receive. This makes them useful for locating moving sources of infrared emissions such as moving people. If we want our robot to follow people around it would be useful if it could remember the direction in which it last detected a person. This way it could still have a chance of moving in the right direction even if the person it was following was no longer moving with respect to the robot.

There are two types of internal state: persistent and ephemeral. Persistent state retains its value until it is deliberately changed. Ephemeral state retains its value only for a specified period of real time and then automatically reverts to a default value. Note that the distinction between persistent and ephemeral state runs orthogonally to the distinction between short-term and long-term state. Persistent state can be short-term if it is changed quickly. Likewise, ephemeral state can be long-term if it is given a long duration.

The use of persistent internal state in the control of primitive activities has been the subject of some debate. Connell has argued that persistent internal state should be avoided altogether [Connell89]. The main reason for this is that if the value of a state variable is based on an erroneous sensor reading then storing that value as internal state will cause that error to affect the robot's actions for longer than it would otherwise. The use of ephemeral state guarantees that no single error will persist for longer than some fixed time period.

However, there are situations where persistent state can be extremely useful. For example, if our robot is following the left wall and decides to take a detour from the wall in order to avoid an obstacle, we would like it to remember which way it was going when it returns to the wall. Although this state is typically short-term, it cannot be ephemeral because we do not know ahead of time just how long it will take the robot to return to the wall. On the other hand, if something goes wrong and the robot wanders off someplace where it can no longer find the wall, we do not want it to continue searching forever. At some point the robot should realize that something has gone wrong. However, it should not simply forget that it was following a wall; it should realize that it has spent too long looking for the wall without finding it. To do this it needs a piece of persistent state to tell it that it was following a wall to begin with, as well as a piece of ephemeral state to tell it when to stop searching.

All of these arguments are rendered somewhat moot by the observation that ephemeral state can be implemented in terms of persistent state (assuming a real-time clock is

available), and vice versa. My system therefore provides both. Persistent state is provided as a primitive, and ephemeral state is provided as a syntactic extension. (See section 3.1.6 and 3.1.8).

Theoretically we can construct a state variable using only a prioritized channel with two inputs. If we connect the low-priority input to the channel's output then the channel will maintain its last value whenever its high-priority input is not being driven. (See figure 2-5a.) However, this solution is cumbersome from a software point of view, and the resulting hardware circuit would not actually work in practice. (The deviations of the performance of a real operational amplifier from the ideal model would cause the output to be unstable when the driving input was removed.)

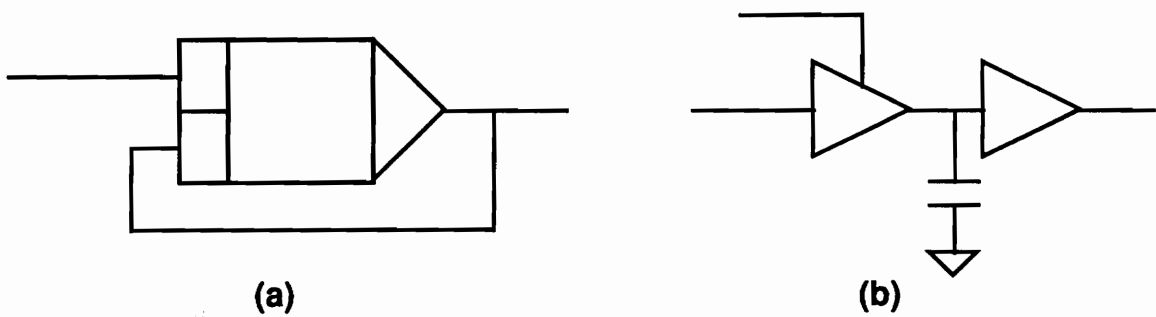


Figure 2-5: (a) A state variable implemented with a prioritized channel. (b) A sample-and-hold amplifier.

In order to reliably store state information in real hardware we can use a device called a sample-and-hold amplifier which employs a capacitor in order to stabilize the circuit. (See figure 2-5b.) Because a different device is used to implement them, and because it is much more straightforward from a programming point of view, state variables are implemented as a separate language construct rather than building them out of channels.

The value of a state variable is set using the `SET` command, e.g.

```
(SET VARIABLE-1 [expression])                                    (2-7)
```

A `SET` command is semantically identical to a `DRIVE` command in the same sense that a state variable is identical to a channel. A different command is used in order to reinforce the difference between the two actions, namely, that the value of a state variable persists

after it is no longer being actively `SET`, whereas the value of a channel is affected by a `DRIVE` command only as long as the `DRIVE` command is activated. Thus, `SET` is typically useful only within the context of a conditional activation.

State variables must be used with great care. Aside from the problem of allowing sensor errors to persist, they can also cause difficulty if feedback loops are introduced. For example, it is sometimes tempting to write something like:

```
(IF condition (SET VAR1 (+ VAR1 1)))                              (2-8)
```

but this will probably not do what the designer intended because this statement is a description of an analog circuit in a feedback configuration - one which happens to be unstable - and not a sequential computation. Expression 2-8 will cause `VAR1` to be incremented continuously for as long as `condition` is true, which may or may not be what was wanted. In order to count discrete events such as transitions of a sensor from one value to another one must do something like:

```
(IF (NOT (= SENSOR OLD-VALUE))                                    (2-9)
  (BLOCK
    (SET COUNT (+ COUNT 1))
    (SET OLD-VALUE SENSOR)))
```

Clearly, doing anything resembling a classical sequential computation is quite cumbersome. This is intentional. The proper use of state variables is to store the values of sensors to make those values available after the sensor no longer provides them, and not to support sequential computations. (We will open that Pandora's box in the next section.)

2.1.5 The Language

The computational infrastructure described above has been implemented as a programming language called ALFA (A Language For Action). ALFA is described in detail in the next chapter. In order to support the discussion of the rest of the architecture, the key features of the language are briefly described here.

ALFA is designed to describe networks of analog circuits which compute highly nonlinear transfer functions from large numbers of inputs to large numbers of outputs.

ALFA programs are networks of modules and channels which are created using the following top-level forms:

```
(DEFINE-CHANNEL name input-spec { :ACTUATOR interface } )

(DEFINE-MODULE name command* )
```

The `input-spec` argument to the `DEFINE-CHANNEL` command specifies the type of the channel, how it is to perform conflict resolution, or if it is to be connected to an external sensor. The `:ACTUATOR` keyword specifies that the output of the channel is to be made available external to the ALFA program.

The two fundamental commands used to define modules are the `DRIVE` and `SET` commands described in the preceding discussion. ALFA supports conditional expressions and conditional activation. In addition, the language has mechanisms for syntactic abstraction and type abstraction. It also has special data types for creating state variables whose values are synchronized to a real time clock. All of these constructs are described in detail in Chapter 3.

A key point about ALFA is that it has extremely impoverished capabilities for dealing with internal state and performing traditional sequential computations. ALFA is a dataflow language. All commands in ALFA run in parallel. There is no sequential execution and no loops. These capabilities can be emulated through careful use of state variables, but the language was specifically designed to discourage this. The reasons for this are discussed in the next section.

2.2 The Sequencing Layer

ALFA and similar systems have been used to program robots to carry out some very complex tasks (e.g. [Connell89], [Mataric90] and Chapter 4 of this thesis). However, there is a practical limit to what can be done with such simple computations. To get around this limit we must introduce some very different sorts of computational mechanisms. This section describes the second layer of the ATLANTIS[2] architecture which provides computational mechanisms for controlling sequences of activities.

---

[2]All The Layers Are Not The Identical Structure

2.2.1 The Problem

Consider a robot equipped with ultrasonic range sensors, some crude odometry, and perhaps some contact sensors, but no vision capability, and imagine that we wish to make this robot go to the second office on the left. Using ALFA we program a set of transfer functions which let the robot align itself to a wall and travel along it. This done, it might seem that taking the second left would be a trivial task: the robot needs only to follow the wall until the range to the wall suddenly increases indicating the presence of the first door. The robot then passes the first door, sets a state variable to indicate that it has done so, and repeats the same procedure. Having found the second door, it turns and goes through and that is that.

Unfortunately, such an approach would work only under the most ideal of circumstances. Suppose, for example, that the door to the first office on the left was closed and thus invisible to the robot's sensors (assuming that the robot is equipped only with sonars or infrared proximity sensors). A robot based on the simple scheme described above would blithely drive into the third door on the left. What if the second door were closed? What if all of the doors along the hall happened to be closed - what would the robot do upon arriving at the end of the hall? What if someone leaves a box on the floor by the wall between the first and second door? What if specular reflections caused the sonars to register a door where in fact there was none?

To any one of these contingencies one could easily devise a minor modification which would get around that particular problem. Perhaps even to any two or three. But a system that can deal with them all, and perhaps even others which the designer never forsaw, is no longer so straightforward. The case for taking the second left is, perhaps, somewhat debatable. A clever programmer might be able to produce an extremely robust second-left taker in ALFA. However, at some point the complexity of the programming required becomes unmanageable, and a better approach must be found. Certainly one would not want to program a robot to go to Hoboken using nothing but ALFA.

Higher-level activities such as going to Hoboken and taking the second left involve generating sequences of primitive activities. In the case of taking the second left the sequence can be quite short and straightforward: follow the hall, pass a door, follow the hall some more, turn left. When contingencies arise, the required sequence changes:

follow the hall, go around the obstacle, follow the hall some more, turn around and go back because you've gone too far, etc.

As we observed in the previous section, dealing with contingencies often requires the use of persistent internal state. To reiterate the point, consider our second-left-taking robot when faced with the decision of what to do when it encounters a door. It must decide whether to go past this door or whether to turn. Assuming that there are no detectable features that would allow the robot to distinguish the second door on the left from any other door on the hall, the only way the robot can make this decision is to remember what has happened to it in the past: is this the first door it encountered since embarking on its mission or the second?

Dealing properly with contingencies generally requires the use of more and more state information in more and more sophisticated ways. For example, one way to double-check the door count is for the robot to measure roughly how far it travelled and compare this with an internal a-priori map of the hall. Performing this measurement requires that the robot note the reading on its odometer when it began the journey. If the robot has to alter its course to avoid an obstacle, it needs to remember that it was following the left wall and not the right in order to resume its task once the obstacle has been bypassed.

ALFA is not suitable for generating such robust sequences of activities because it was specifically designed to have impoverished capabilities for dealing with persistent internal state. This is because the use of such state is generally a bad idea in the control of primitive activities. However, in the control of higher-level activities it is crucial. Therefore we must develop a computational structure which will allow us to manage persistent internal state of potentially great complexity.

2.2.2 Cognizant Failure

The main job of the sequencing layer is to recover when things go wrong (as they inevitably will). In order to do this, it first has to be able to tell when something has gone wrong. If a failure goes undetected then the robot has little hope of recovering from it except by luck.

This simple principle has extremely broad implications for the design of a mobile robot. It reaches all the way down to the design of the robot hardware, where it dictates that the robot must be equipped with sufficient sensors to detect failures.

Let us refer to a failure that can be detected by the robot as a cognizant failure, because the robot can tell that it has occurred. Designing a robot from the ground up to fail cognizantly is absolutely crucial to robust intelligent behavior. There are two reasons for this. First, we want to be able to build robots that can potentially deal with situations which we have not explicitly anticipated in their design. Our robot may have perfect dead reckoning and a provably correct path planning algorithm, but when building maintenance installs carpeting where there was once tile, such a robot will fail. It is preferable to have a robot that will stop and report that something unexpected has occurred than one which blithely slams into the wall.

The second reason for pursuing cognizant failure is that it is much easier to develop algorithms that fail cognizantly than ones which never fail. (It is not at all clear that it is even possible to design an algorithm to control a real robot that never fails.) Furthermore, by detecting failures and invoking recovery procedures, unreliable algorithms can be combined into an overall algorithm which is far more reliable than any of its component algorithms [Firby89]. It is this sort of combination which is performed by the sequencing layer of the ATLANTIS[3] architecture.

2.2.3 Interfacing to Primitive Activities

We begin by describing the interface between the sequencing layer and the control layer. The sequencing layer communicates with the control layer in three ways. First, it can initiate or terminate a primitive activity by enabling or disabling the ALFA modules which control that activity. Second, it can fine-tune the behavior of the primitive activity by sending parameters to that activity through ALFA channels. Third, it can monitor the status of the activity by examining the values of the control layer's external channels. These channels may include raw or processed sensor data, information about the internal state of a module, or actuator commands.

---

[3]Architecture To Lend Autonomous Navigation To Intelligent Systems

For each primitive activity implemented in the control layer, there is a corresponding set of three procedures in the sequencing layer called the initiator, the monitor, and the terminator. Using this interface we can construct classical atomic Actions (with a capital A) simply by calling the initiator, calling the monitor until the desired effect is achieved, and finally calling the terminator. However, we can also do more sophisticated things.

For example, we can invoke two different primitive activities at the same time. So, for example, using our interface we can move the robot and aim the camera at the same time. This is beyond the capabilities of the classical discrete-action model. We can also initiate or terminate an activity in the middle of another activity. Our robot could aim its camera multiple times while wandering down the hall.

This capability comes at a cost, however. We must be careful not to allow two activities which interfere with each other to be active at the same time. If our robot has a manipulator arm, the robot probably needs to be stopped before it can be used to pick up an object. This can be a significant problem when unexpected situations arise. If we wish to initiate an activity to recover from some contingency we must make sure to terminate all of the activities with which the recovery activity might interfere. Sometimes this can be quite complicated, as we shall later see.

Insuring that no conflicts occur requires careful engineering. However, there should be a mechanism to guard against inadvertent activation of two interfering primitives, since this could cause damage to the robot. To accomplish this, each robot resource has a semaphore associated with it in the sequencing layer. An initiator for a primitive activity must first acquire this semaphore before it initiates its activity. If the semaphore is not available, the initiation fails.

For example, consider a robot with primitives for aiming its camera, moving, and picking up an object, and suppose that this latter activity requires that the robot be stationary. The initiator for the moving primitive and the object-collecting primitive would both take the semaphore associated with the robot's drive motors. Thus, if an erroneous task schema attempted to activate the object-collecting primitive while the robot was moving, the initiation would fail (cognizantly!). Likewise, the robot would be prevented from driving away before it was through collecting an object.

There are times, of course, when it is necessary to abort one activity in order to initiate another, higher-priority activity which requires a resource held by the first activity. This

turns out to be quite a difficult problem, and not unique to primitives. It is discussed in section 2.2.6.

2.2.4 Tasks Schemas, Tasks, and the Task Bin

The next step is to develop the computational structure needed to manage the activation and deactivation of primitive activities. We need a convenient way to specify event-driven conditional procedures, possibly with very high branching factors, that will achieve the robot's goals. In particular, we need to be able to tell the robot what to do when things go wrong without having to specify exactly what to do in every possible situation.

The solution borrows heavily from Firby's Reactive Action Package (RAP) system. In fact, the ATLANTIS sequencing layer is essentially a RAP interpreter modified to control activities rather than discrete actions. Some terminological changes have been made as well. A complete discussion of the relation of ATLANTIS and RAPs appears in Chapter 4. For now we will briefly describe the control structure as implemented in ATLANTIS. For further details the reader is referred to [Firby89].

The heart of the sequencing layer is a data structure called a task schema[4]. A task schema is a collection of methods for accomplishing something, together with annotations describing under what circumstances each method is applicable. For example, a task schema for following walls might contain two methods, one for when the initial orientation of the robot relative to the wall is known, and the other for when it is not. The first method might contain three steps: initiate a wall-following activity in the control layer, monitor that activity until it completes or fails, and finally terminate that activity. The second method would begin with a step to align the robot to the wall (which would in turn initiate, monitor, and terminate some sort of alignment activity).

Methods may do one of four things: they may initiate, terminate, and monitor primitive activities, and they can invoke other task schemas. When a task schema is invoked it

---

[4]A task schema is the same as a RAP. The term *task schema* is, hopefully, somewhat more descriptive.

becomes a task and is inserted into a task bin[5]. The task bin is just a list of the tasks in which the system is currently engaged.

The task bin provides a convenient way of dealing with the failure of an activity. Whenever a primitive activity fails, the failure is reported to the task which initiated it[6]. This task then attempts to recover from the failure, either by trying the activity again, or by choosing an alternate method for achieving the task's goal. When all of a task's methods have been tried and failed, the failure is reported to the task which invoked it. This way every alternate method known to the system for achieving a goal is tried, resulting in robust execution even in the face of high failure rates. Firby's work developed a number of powerful heuristics for choosing among alternate methods based on what is known about the state of the world.

2.2.5 Task Sequencing

Deciding what task should be executed at any given point in time is a fairly complex problem which Firby addresses extensively. I will review his work briefly here. To distinguish between Firby's system and the extensions made by ATLANTIS, I will use Firby's nomenclature when discussing his work.

Each RAP, in addition to its methods, contains annotations which tell the sequencer when it should be run. These annotations include a set of preconditions, which specify conditions which must be true before the task may run; a set of constraints, which are preconditions that get inherited by a task's sub-tasks; and a priority, which is used to choose among mutually-exclusive tasks which are otherwise eligible to run. High priority tasks can interrupt low-priority ones before they are finished.

RAPs are executed by an interpreter which operates in cycles. Each cycle consists of two phases. In the first phase, one of the RAPs in the task queue is chosen based on a set of heuristics. In the second phase, one of the RAP's methods is chosen based on the method annotations and the current perception of the state of the world. A method is either

---

[5]The task bin is the same as the RAP task queue. The terminology was changed because the tasks in the bin are only partially ordered, and thus it is not appropriate to call this data structure a queue.

[6]The failure can also be reported to the deliberative layer. This will be described in section 2.3.

a primitive, atomic Action, in which case it is simply executed, or it is a set of RAPs, in which case these are inserted in the task queue. The cycle then repeats. Because a given RAP only controls the system for a single cycle at a time, RAPs can respond quickly to both unexpected failures and opportunities.

The power of RAPs comes at least as much from the coding of the RAPs as from the underlying system. Unfortunately, writing RAPs is hard, at least as hard as programming in general. In some ways it is more difficult to write a RAP than a traditional program because the robot can interact with the environment in unexpected ways. While this might be improved some day, for now writing task schemas is still mostly an *ad hoc* process. However, an important result of Firby's work is that writing RAPs for controlling high-level activities requires less effort than for low-level activities. This is because the low-level RAPs, if they are properly designed, tend to hide many of the ugly details of interacting with the environment. Writing low-level RAPs is difficult, but it only has to be done once (for a given set of hardware). The situation is very similar to writing a device driver for a traditional operating system. Device drivers are usually complex, *ad hoc* pieces of code that are quite difficult to write and maintain. However, once they are written it is very easy to write applications software because all of the ugly details of dealing with the real world (disk drives, networks, etc.) is now hidden by the driver. ATLANTIS has little to add to the methodology of writing task schemas. It is still a black art, but it only has to be done once.

There are some minor changes needed to the way in which ATLANTIS task schemas are executed by the system. These are caused by the fact that primitive activities may execute simultaneously, whereas RAP primitives could only be invoked one at a time. The RAP system choses a single task to execute or expand at each cycle; the ATLANTIS sequencer can execute multiple tasks. At each cycle, all of the eligible tasks (i.e. those whose preconditions are true) are collected and their methods executed in order of priority. A high-priority task may seize exclusive control of the task bin (if necessary) by taking a resource semaphore which other tasks require in order to run.

2.2.6 The Wesson Oil Problem

A more difficult problem arises when a high-priority task attempts to initiate an activity whose resources are currently in use by a low-priority task, a problem I call the Wesson Oil

problem. The name derives from a television commercial for Wesson Oil in which a housewife is frying chicken (in Wesson Oil, of course) when one of her children suddenly falls down and has to be taken to the hospital. However, before going to the hospital the housewife turns off the stove. When she returns an hour later she resumes frying the chicken (which turns out crispy despite the fact that it has been soaking in the oil for an hour).

The action of turning off the stove is very interesting because the RAP model of execution cannot account for it. It is an example of a *clean-up procedure* which is executed when a high-priority task (taking the kid to the hospital) interrupts a low-priority task (frying chicken). Turning off the stove is done as a direct result of the hospital task taking control of the task queue, and yet turning off the stove is not normally part of going to the hospital. It is part of the chicken-frying task which gets executed before that task gives up control to another task.

In a similar fashion, if a high priority task needs to initiate a primitive activity which would interfere with a primitive initiated by a low-priority task, then the low-priority task must be given an opportunity to execute a clean-up procedure before the high-priority task takes control.

One solution to this problem is to augment the sequencer with an "unwind-protect" feature. Unwind-protect is a LISP construct which "protects" the execution of a program by executing a clean-up procedure after the program is done even if the program was interrupted or produced a run-time error. For example, unwind-protect can be used to emulate dynamic binding in a system which provides only lexical binding as a primitive. A clean-up procedure to return the dynamically bound variable to its original value guarantees that the value will be unchanged even if the program within the scope of the binding terminates abnormally.

This is only a partial solution to the problem. Ideally, it should be possible to make the execution of the clean-up procedure be conditional upon other things. If a stranger walks into your kitchen with a gun, turning off the stove before running for your life may or may not be the right thing to do. This can be an extremely complicated problem, requiring information about the urgency of the interrupting task, as well as information about the time and resources that it is expected to consume. Predicting such things can get to be a very thorny problem.

2.2.7 Real Time and Caching Computations

The sequencing layer does not need to respond to situations as fast as the control layer because the control layer (presumably) keeps the robot out of trouble while the sequencing layer is deciding what to do next. However, the sequencing layer does need to respond as quickly as possible to situations, otherwise opportunities for efficient action might be missed. For example, there is a limited amount of time available between sighting a sign for a gas station on the freeway and arriving at the point where the decision to exit must be made. If the decision is not made in time it is not necessarily an immediate disaster (as a collision would be) but it might result in a very inefficient overall course of action (involving, say, a walking trip to the gas station which might otherwise have been unnecessary).

One way to help the sequencing layer run faster is to provide a mechanism for caching the results of computations which would otherwise have to be performed several times. For example, later we will encounter a robot which navigates through outdoor natural terrain. Along the way it periodically computes a direction vector which indicates the preferred direction of travel at the robot's current location. This computation can take several seconds, and must be performed by two different tasks: one which controls the robot's direction and another which controls the direction in which the robot's cameras are pointed. These computations can be made more efficient by adding a separate task to compute the direction vector, the result of which is used by both the navigation and camera-aiming tasks.

ATLANTIS accomplishes this by structuring time-consuming computations performed by the sequencing layer as cachable function calls. A cachable function call is one which stores its result in a permanent state variable the first time it is called. Subsequent calls simply return the value of that variable, thus avoiding the time required to actually recompute the value. A cachable function can be forced to recompute its value by resetting the function, an operation which clears the previous result and forces it to be recomputed on its next invocation. Alternatively, cachable functions can be set up which automatically reset themselves based on external conditions such as the passage of a given period of time since the last recomputation, or a significant change in the value of a sensor.

Cachable functions are defined using the following form:

```
(define-cachable (name . arguments) recompute-test . body)
```

When a cachable function is called, it evaluates the recompute test first. If the recompute-test returns false, the function returns the last cached value. Otherwise, the body of the function is evaluated and the result is stored. (cf. the WHEN-CHANGED construct in [Miller85])

There are two special forms which may be used in the specification of the recompute function. They are:

```
(changed? expression)
(elapsed-time)
```

The CHANGED? special form returns T when the value of expression is different from what it was the last time that the cachable function was recomputed. The ELAPSED-TIME special form returns the time in seconds since the last time the cached function was recomputed. These special forms may be combined with each other and with external function calls by the boolean connectives AND, OR and NOT to form the compute test. For example, a cached function which recomputes its value under any of three circumstances -- a changed argument, an elapsed time of more than ten seconds, or an external condition called RECOMPUTE-FOO? -- may be specified as follows:

```
(DEFINE-CACHABLE (foo x) (or (changed? x)
                             (> (elapsed-time) 10.0)
                             recompute-foo?)
  {body of function} )
```

2.3 The Deliberative Layer

We have described the mechanism by which primitive activities are controlled, and the mechanism by which primitives are initiated and terminated. We must now address the problem of how to support time-consuming deliberative computations. Such computations cannot be allowed to take place directly in the sequencing layer or its response time may be severely compromised. Instead, a separate layer, running asynchronously from the rest of the system, performs time-consuming deliberative computations in ATLANTIS[7].

---

[7]A Type of Layered Architecture Needed To Integrate Strata

Historically there have been two kinds of time-consuming computations associated with mobile robots: sensor processing, especially vision processing, and planning. Planning is inherently time-consuming, while vision processing can potentially be made very fast, though current technology is not yet up to the task.

We have an existence proof that fast vision is possible in human beings. The human visual system is a truly remarkable piece of computational machinery. It is entirely passive (quantum mechanical weirdness notwithstanding). It provides information about the environment at high bandwidths (by comparison to the mechanical bandwidth of human muscles) and with exceptional fidelity. Furthermore, the information is processed until it can be presented to our conscious thought processes at a very high level of abstraction. We look at the world and we see objects, despite the fact that the raw data perceived by the retina is a field of irregularly spaced pixels.

At present, no artificial vision system can match this ability. The state of the art in vision processing requires calculations which are quite time-consuming. The amount of time required to process a vision frame can be quite significant in comparison to the amount of time available for a robot to respond to an event. Therefore, if we are to incorporate vision into our robot we must do so in a way that the time spent processing vision data does not adversely affect the overall performance of our robot.

Humans occasionally perform inherently time-consuming computations as well, though typically not connected to visual processing. We will sit down with a map and plan a route to Hoboken. We might spend time thinking about the best way to avoid a traffic jam. We will even puzzle at length about where we should stop to eat, occasionally taking so much time doing it that by the time the computation is complete the opportunity to act on the outcome has passed.

From the point of view of a robot control system there is very little difference between time-consuming sensory processing and planning. Both are time-consuming and resource-consuming processes which yield some more-or-less useful information upon their completion. Thus, from an architectural point of view we can treat these computations as essentially identical.

There is a vast literature on deliberative algorithms for mobile robot control, and ATLANTIS has little of interest to add to this body of knowledge. (But see Chapter 5 for some useful hacks.) What ATLANTIS does take a stand on is how those computations

should interface to the rest of the system. There are two questions which must be answered. First, what determines how the deliberative layer will allocate its scarce computational resources? Second, how do the results of the deliberative layer's computations get used by the rest of the system?

There are two possible answers to the first question. The deliberative layer might contain within it the means by which to choose what it should compute next. In other words, the deliberative layer monitors the robot like an overseer and decides on its own what sort of information the rest of the system needs most. The second possibility is that the sequencing layer initiates and terminates computational processes in the deliberative layer the same way that it initiates and terminates computational processes in the control layer.

I will argue for the second method on three grounds. First, computational processes are in some sense isomorphic to physical processes. Second, the first method requires reproducing much of the functionality of the sequencing layer in the deliberative layer. Third, the second method subsumes the first. I will address each of these arguments in turn.

Consider the physical process of aiming a camera mounted on a pan-tilt head into the proper position to acquire an image. This is a time-consuming and resource-consuming process performed for the purpose of acquiring information. Some of the information in the image might be usable directly by the sequencing layer, or even the control layer. For example, the existence of a red traffic light in the visual field could conceivably be determined very quickly by a matched-filter algorithm [Wehner87] or by a visual routine [Ullman84]. On the other hand, detecting arbitrary obstacles in the visual field is computationally demanding given current technology. To get this information a computational process must be initiated. This process, like aiming the camera, is a time-consuming and resource-consuming process performed for the purpose of acquiring information (or, more precisely, to transform existing information into a usable form). From the point of view of the rest of the system, therefore, there is no difference between a deliberative computation and aiming a sensor.

One key similarity between computational and physical processes is that both must be interruptible. This is necessary for three reasons. First, the world may change in a way which makes the result of a computation obsolete before it is finished. Second, it is

possible to initiate computational activities which will never terminate on their own. We do not want our robots to go permanently catatonic if for some reason they ever initiate a non-terminating computation[8]. Third, it may be necessary to suspend a computational process to deal with an urgent contingency if the contingency requires some deliberative computations. In Chapter 5 we will see a robot which engages in global path planning while it is travelling. Occasionally this robot gets stuck and needs to plan a short path to extricate itself. This robot must be able to suspend the very time-consuming global planning in order to plan the short recovery path that it needs to be on its way.

It is very straightforward to control this sort of thing from the control layer; all of the mechanisms required are already there. To do the same thing from the deliberative layer it would have to continuously monitor the situation in the world and continually reassess whether or not its current computation is worthwhile, or if it needs to start thinking about something else.

Finally, controlling computations from the sequencing layer does not preclude the possibility of the deliberative layer making its own decisions about what to think about. The sequencing layer could initiate a computation in the deliberative layer which begins by choosing what to compute. This can occasionally be useful in cases where the sequencing layer is at a loss for something to do. Usually, however, if the sequencing layer's task schemas have been well designed, they will tell the sequencing layer how to decide what deliberative computations are most urgently needed.

Finally, we must address the problem of how to present the results of the deliberative layer's computations to the sequencing layer. The sort of information which passes from the deliberative to the sequencing layer is quite different from that passed from the control layer. The control layer only produces scalar values or arrays of scalar values. Deliberative computations, on the other hand, commonly produce complex linked data structures as their output. This is not really a problem, but it does mean that to implement

---

[8]This was the basis for an episode of the popular science fiction television series *Star Trek* in which an alien which had taken over control of the Enterprise's computer was finally evicted by ordering the computer to calculate π to the last decimal place. The existence of subsequent episodes implies that the computer was equipped with a mechanism by which the calculation could be aborted once the alien was safely banished.

this interface requires something more complex than a channel -- a shared-memory data base or a blackboard system, for example. The structure of the data base is highly dependent on the design of the robot, the tasks it is to perform, the sensors and actuators with which it is equipped, and other factors which vary from instance to instance. Thus, the internal structure of the deliberative layer and the data base will be left unspecified. Chapter 5 presents an example of an implemented deliberative layer running on a real robot.

2.4 ATLANTIS: A Summary

This chapter described an architecture for a control system for a mobile robot consisting of three components: a control layer, a sequencing layer, and a deliberative layer.

2.4.1 The Control Layer

The control layer is a (mostly) stateless reactive control system. It is fundamentally a traditional analog feedback control mechanism. However, the transfer functions that it computes tend to be highly nonlinear vector functions of high dimensionality. In order to construct these functions in a reasonable way we have devised a programming language, ALFA, for building up such functions in a modular way. The semantics of ALFA are based on analog circuits, but the language is usually compiled to uniprocessor code which simulates the circuit at some finite sampling rate.

ALFA permits modular design through the use of communications channels. Channels are computational entities which combine an arbitrary number of inputs into a single output. All communication in ALFA, whether internal or external, takes place through channels.

The computational components of ALFA are called modules. All input to a module comes from the outputs of channels and all output from a module goes to the inputs of channels. Connections to channels are made by referring to the channel by name within the definition of a module. This makes it possible to insert and remove modules without having to restructure the communications network.

Where prioritized communications are used, the priorities are set up within a channel definition. Thus it is possible to establish communications paths within an ALFA program which cannot be overridden by adding code, only by changing the channel definition. This allows one to structure code in a way that one can be confident that the addition of new functionality will not interfere with existing abilities.

2.4.2 The Sequencing Layer

The sequencing layer controls activities which consist of temporal sequences of primitive activities. The sequencer is essentially the same as Firby's Reactive Action Package (RAP) system. However, the fact that activities can go on simultaneously requires some changes to be made.

Like the RAP system, the sequencer consists of a task bin which contains a number of tasks. Each task contains a list of methods for achieving that task's goals, along with annotations specifying under what circumstances the method should be used. When a task is run it can do one of four things: it can initiate or terminate a primitive activity, it can monitor the progress of a primitive activity, or it can insert a new task into the task bin.

Central to the functioning of the sequencing layer is the notion of cognizant failure. Rather than attempt to design algorithms which never fail, we instead design algorithms which can detect failures when they occur. When a task fails, it invokes another of its methods to attempt to recover from the failure. When all of the methods have been tried without success, the failure is propagated up to the task which installed the failed task. The result is a robust system despite the unreliability of primitives.

To prevent two primitive activities which interfere with each other from being activated at the same time, each primitive is annotated with a list of resources it uses. A set of semaphores prevents two interfering primitives from being active at the same time.

This restriction causes a problem when a high-priority activity must interrupt a lower priority activity. In many cases, the lower priority activity must execute a clean-up procedure before the high-priority activity may safely proceed. A partial solution to this problem is to install an unwind-protect mechanism. A general solution to this problem is an open research issue.

2.4.3 The Deliberative Layer

The deliberative layer performs time-consuming computations such as vision processing and planning. Computations in the deliberative layer are initiated and terminated by the sequencing layer in the same way that physical activities are initiated and terminated in the control layer. The results of deliberative computations can be considered low-bandwidth sensor information. This information is usually in the form of complex linked data structures, and so must be communicated through a data base rather than ALFA channels.

There is a vast literature on deliberative computations for mobile robot control. ATLANTIS has little of interest to add to this body of knowledge.

2.4.4 Who's in Charge?

One common feature of new architectures which has been conspicuously missing from this discussion is a block diagram. I have been consciously avoiding drawing a block diagram because they are too often abused. Papers on architectures often put too much emphasis on block diagrams and too little on a description of the architecture, to the point where a naive graduate student (I won't mention any names) can almost get the impression that a bunch of boxes and arrows *is* an architecture[9]. For this reason I have put off the block diagram for as long as possible so that it could not serve as a crutch for the real task which is describing the components of the architecture and the interfaces between them.

However, the time has come to draw boxes, and for the following reason. It is tempting to consider ATLANTIS in terms of a traditional hierarchical design and say that the deliberative layer is "controlling" the robot and that the sequencing layer and the control layer are merely acting in service of this "topmost" layer. It is equally tempting to assign a correspondence between the hierarchy of activities developed in Chapter 1 and the architecture, i.e. to consider that the deliberative layer controls high-level activities and that the sequencing layer control mid-level activities. This view is made even more appealing by the fact that the control layer does, in fact, control primitive activities which form the base of the hierarchy.

---

[9]As Mark Drummond puts it, "Boxes! We got boxes!"

Nevertheless, both of these views of ATLANTIS are incorrect and misleading. *Activities are spread out throughout the architecture.* The activity hierarchy is evident in the structure of task schemas, not in the computational topology of the architecture.

To clarify this, recall how we defined activities and decisions in Chapter 1. An activity is a set of physical and computational processes. Activities are initiated by decisions which are the result of a computational processes which are part of higher-level activities. The function of the ATLANTIS control layer is to control primitive activities, that is, activities with no decision-making computations. The function of the deliberative layer is to control the time-consuming deliberative computations which are required to make high-level decisions effectively. The decisions themselves are all made in the sequencing layer.

This point is worth reiterating: the function of the ATLANTIS sequencing layer is to make decisions, that is, to initiate activities. These activities may be primitives, which are initiated by activating the proper modules in the control layer, or they may be higher-level activities which are initiated by installing tasks in the task bin. The computations which support higher-level decision making are performed asynchronously by the deliberative layer, but all decisions are ultimately made in the sequencing layer.

Figure 2-7 is an attempt to draw a block diagram of ATLANTIS[10]. The control layer is on top not because it is in charge, but because something had to be on top, and putting the deliberative layer on top would reinforce the prejudice that the deliberative layer is controlling everything. None of the components is in charge. The system as a unified whole is in charge.

---

[10]A Tight Little Architecture: Neat Topology, Intelligent System

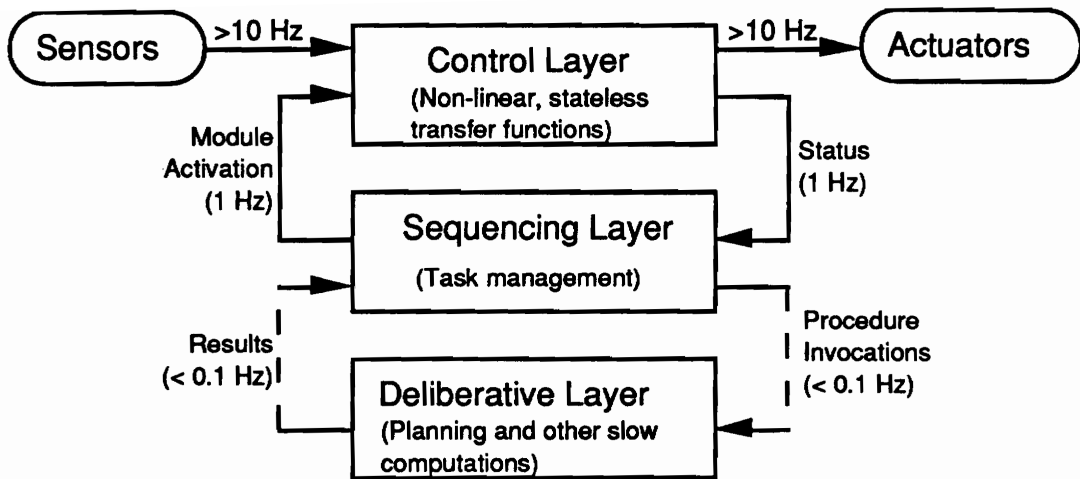


Figure 2-7: The ATLANTIS control architecture.

One might wonder, then, where the system gets its goals to begin with. This is a tricky question because there is a huge variety of information that the system might be given which could be considered "goals". The quick answer is that goals are implicitly located throughout the system. Low-level goals, such as avoiding collisions, are implicit in the structure of the control layer. Higher-level goals, such as a list of destinations for the robot to visit, are implemented as tasks in the task bin. Meta-goals, such as obeying commands given at run time, can be implemented as deliberative processes which interpret the commands and produce new tasks to be installed on the task queue. General advice about the state of the world can be inserted into the system data base. We will see examples of all of these in Chapter 5.

Chapter 3: A Language For Action

The computational infrastructure for supporting the ATLANTIS[1] control layer was implemented as a programming language called ALFA. The reason for designing a new language is that it allows us to explicitly support the sort of dataflow computations, modularization and abstraction described in the previous Chapter. Furthermore, by carefully designing the syntax of the language we can guarantee that any legal program will have a response time with a constant upper bound that can be determined at compile time. We can also eliminate many language constructs designed to support sequential computations. This allows the language to be very simple, and thus easy to compile into very small code which needs very little runtime support. (The latter part of this Chapter will describe an ALFA program that was used to control a mobile robot using less the 5000 bytes of memory.)

3.1 Data Types

It is very important when programming robots that as many errors as possible be detected at compile time. Debugging a robot is much more difficult than debugging a traditional program. When a traditional program crashes it prints an error message on the screen, or, at worst, requires that the computer be rebooted. When a robot program crashes, the robot sometimes crashes into the wall. Furthermore, there is typically no "stack trace" to help the designer figure out what happened.

One of the most powerful tools for detecting programming errors at compile time is strong-typing. In a strongly-typed language the data type of every expression in the program (though not the value) can be determined by the compiler. This allows a large class of programming errors to be detected without having to run the program. However, we want the semantics of ALFA to describe a computation performed by an analog circuit, and it is a little unclear exactly what is meant by "data type" in this context.

In traditional programming terminology a data type is a set of values for which certain operations are defined. For example, the data type BOOLEAN is defined as the set

[1]ATLANTIS: This Little Acronym Needs To Indicate Something

{TRUE,FALSE}. There are a number of things that you can do with a datum of type BOOLEAN. For example, you can negate it (which yields another datum of the same type). You can compare it to another BOOLEAN datum. Or (the most common usage) you can use it as the discriminant of a conditional.

There is an analogous[2] phenomenon in our circuits. Recall the following examples from the previous Chapter:

```
(DRIVE THE-ACTUATOR                                     (3-1)
       (IF (> THE-SENSOR 0.1)
           (* 10.0 (- THE-SENSOR 0.1))
           0.0))
```

and:

```
(IF (> THE-SENSOR 0.1)                                  (3-2)
    (DRIVE THE-ACTUATOR (* 10.0 (- THE-SENSOR 0.1))
    (DRIVE THE-ACTUATOR 0.0))
```

We modelled the expression (> THE-SENSOR 0.1) as an analog comparator which produced a digital output. This output is the circuit equivalent of a BOOLEAN data type. It can take on one of two values, ON or OFF. It can be inverted, and it can be used as the discriminant of a conditional by using it to gate a relay or an analog switch (i.e. a pair of tri-state buffers).

We could also extend our circuit paradigm by allowing combinatorial digital circuits to appear in our devices. Instead of a "wire" we would have a "data bus" to move values of this type back and forth. We can "convert between types" using analog-to-digital and digital-to-analog converters. We can have "arrays" by allowing multiple instances of wires or data busses to be referred to as a unit. Nearly every aspect of traditional data structures in programming languages has a counterpart in circuitry.

This should not come as too great a surprise, since computers are ultimately analog circuits. However, computers are engineered in such a way that all of the analog aspects of the device are hidden from the user. The aim here is to design a programming language that makes the analog nature of the underlying circuits accessible to the user. The

---

[2]No pun intended.

semantics of the language are couched in terms of the stuff that computers are made of, and therefore it is no big surprise that we can recreate computers using it. However, the main point is that we can use some of the abstraction techniques that have been developed for managing complexity in computers and apply them to managing complexity in analog circuits. The fact that by doing this we can recreate digital computers is not very interesting.

In order to preserve the illusion that ALFA is somehow different from, say, Pascal, the primitive data types in the language have been given names which suggest circuits. There are two primitive data types in ALFA: `BIT`, and `ANALOG`. A `BIT` value is what comes out of a comparator - a binary value, on or off, 1 or 0. An `ANALOG` value is a continuous voltage, the sort that you process using operational amplifiers.

There are a number of other types in the language. For example, there is a `BUNDLE` type, which is exactly the same as an array. There is a `DIGITAL` type, which is the same as a `BUNDLE` of `BIT`s. `DIGITAL` is roughly the equivalent of the `INTEGER` type in traditional programming languages.

Thus, it is possible to strongly type ALFA despite the fact that its semantics are grounded in circuits rather than sequential computations. The reason for going into such detail at this point is that strong typing has a major impact on the syntax of the language: it occasionally requires the introduction of type declarations. These declarations will occasionally appear in subsequent examples. The reader is strongly urged not to be fooled by them into thinking in terms of traditional sequential programming languages.

3.2 Modules

Modules are the basic computational unit in ALFA. Grouping computations into modules serves two purposes:

- It allows the programmer to organize computations into related groups in order to simplify program development.
- It permits the driving commands for prioritized channels to have the same syntax as those for driving other types of channels by allowing the priorities to be set within the channel definition. (See the next section.)

The syntax for creating modules is straightforward:

```
(DEFINE-MODULE name command* )
```

The following is a list of all the legal commands:

```
(IF expression command [ command ] )
(COND ( ( expression command* )* )
(CASE expression
        ( { expression | (expression expression) } command* )* )
(SET state-variable expression)
(DRIVE channel-name expression )
(BLOCK command* )
(LET ( ( variable expression )* ) command* )
(SLET ( (variable-name initial-value) ) command* )
(TRIGGER timer-variable)
(ASSERT assertion-variable)
```

COND and CASE are syntactic sugar for a collection of IF's. IF, COND and CASE can all be used to construct conditional expressions as well as for conditional command activation.

CASE allows indices which are either single atomic constants or lists of two atomic constants. If the index is a list of two items, they specify an inclusive range against which to check the key. This allows convenient specification of actions which vary according to the range of a particular sensor value.

BLOCK is used to group a set of commands into one. LET allows assigning local names to the outputs of computational units, allowing them to be referred to more than once without having to be recomputed. In effect it is identical to LET in LISP except that local variables in ALFA may not be SET. Note that the commands within a LET or a BLOCK nominally run in parallel. Even though in practice they usually run sequentially, one should be careful not to write code that relies on this.

SLET is used to set up state variables. SLET, along with TRIGGER and ASSERT will be described in a later section. They are included here only for completeness.

When ALFA is compiled onto a traditional processor, the commands are compiled into a procedure which computes the current value of all the module's outputs. This procedure is called the module's *method*. This term is also used on occasion to refer to the list of commands in the module's definition.

3.3 Channels

Channels are created using the following form:

```
(DEFINE-CHANNEL name input-spec { :ACTUATOR interface } )
```

The input-spec argument specifies the source of the channel's input and how to resolve conflicts among multiple inputs. This input spec is one of the following:

```
:PRIORITY default module-name+
:MAXIMUM floor
:MINIMUM ceiling
:AVERAGE
:SENSOR type interface
:COMPUTED expression
```

The output of a `:PRIORITY` channel is the value at which the channel is being driven by the highest priority module, i.e. the one latest in the list of modules. Only modules in the list may drive the channel. If no module is driving the channel then the output value is the default value.

The output of a `:MAXIMUM` (`:MINIMUM`) channel is the highest (lowest) value at which the channel is being driven. The `floor` (`ceiling`) argument is the lowest (highest) value that the channel may output, and serves as the default value when there are no inputs.

The output of an `:AVERAGE` channel is the instantaneous average of all its currently active inputs. If there are no active inputs, the output is 0.0.

The value of a `:SENSOR` channel is supplied by an external source, usually a physical sensor. The interface argument is a device-dependent specification of the outside interface. For the current ALFA compiler the interface is either a numerical or symbolic memory address at which the value of the channel is deposited by a device driver.

The output of a `:COMPUTED` channel (also called a virtual sensor) is the value of the expression. These channels are simply syntactic sugar for for creating virtual sensors, allowing a module and a "private output channel" to be combined in a single defining form. Computed channels cannot be driven.

Finally, the output of a channel may be made available external to the ALFA program by specifying the optional :ACTUATOR keyword. The interface argument is a device-dependent interface specification. The current ALFA compiler assumes that the interface is a numerical or symbolic memory address at which ALFA is to deposit the value of the actuator for use by an external device driver.

Sensor channels must have their type declared since there is no way for the compiler to infer the type. Sensor channels (and only sensor channels) may be of BUNDLE (i.e. array) types. The types of other channels are inferred from their default values. Averaging channels may only be of type ANALOG (i.e. float).

3.3.1 Channel Examples

The following are examples of channel definitions.

```
(define-channel sensor1 :sensor                              (3-3)
                (:bundle :digital 5 5) "_occ_grid_1")

(define-channel ch1 :maximum 10.0)                           (3-4)

(define-channel motor1 :average :actuator "_motor_1")        (3-5)

(define-channel prioritized-ch :priority :TRUE b1 b2 b3)     (3-6)

(define-channel virtual-sensor1 :computed (sqrt ch1))        (3-7)

(define-channel in&out-chnl :sensor :analog "_in"            (3-8)
                            :actuator "_out")
```

Example 3-3 defines an interface to an external sensor which supplies a 5x5 array of integers. The symbolic address of the array is "_occ_grid_1".

Example 3-4 defines an internal communications channel where conflicts are resolved in favor of the driving signal of greatest magnitude. If no modules drive the channel at a value greater than 10.0, the output will be 10.0. (Another way to think about it is that there is a constant driving signal of 10.0 present at the channel's inputs.)

Example 3-5 defines an interface to an external actuator. Conflicts are resolved by averaging all of the driving signals. In addition to being available to ALFA modules, the value of the channel is sent to a device driver via the external label "_motor_1".

Example 3-6 defines an internal channel whose conflicts are resolved by priority. Behavior B1 has the lowest priority, B2 the next highest, and B3 the highest. No other module may drive this actuator. This channel is of type `BIT`. If none of the three modules drives the channel, its value is :TRUE. (Boolean constants may be specified as any of :TRUE, #T, or :ON, and :FALSE, #F, or :OFF.)

Example 3-7 defines a channel whose value is the square root of the value of the channel `ch1`.

Example 3-8 defines a channel which receives its value from an external source (referenced by the label "_in") and also sends its output to an external destination (referenced by the label "_out").

3.3.2 The implementation of channels

Conceptually, channels are analog hardware. In practice ALFA is compiled into traditional uniprocessor code which simulates the analog circuit. In such a simulation channels are implemented as follows.

Internally, a channel is a data structure with a number of state variables together with a *drive method* and an *update method*. The drive method and the update method are very similar to module methods, the routines which simulate modules, in that they are computational procedures which contain no loops. The drive method is called whenever a module drives the channel. The value at which the channel is driven is passed to the drive method as an argument. The drive method then performs some computation, storing intermediate results in the channel's state variables.

When all module methods have been run, the channel's update method is called. The update method uses the values stored in the channel's state variables to compute the final value for the channel for the next iteration. Prioritized channels work in the same way, except that the priority is passed to the drive method as a second argument.

This general structure was chosen because the computation of a channel's value should conform to the following criteria:

- The computation should require time linear in the number of driving modules.

- The computation should not depend on the order in which the driving modules are processed.

Any mediation scheme which meets the above two criteria can be implemented within the framework of an ALFA channel. (Discrete constraint-propagation channels, for example, can be implemented as ALFA channels). It is also possible to define mediation strategies which do depend upon the order of processing. Such strategies should be avoided since there is no way to tell in what order modules will be processed. This is the reason that new mediation methods may not be defined in an ALFA program but must be implemented as compiler extensions.

3.4 State Variables

State variables are established using the `SLET` (state-let) form. The syntax of SLET is the same as LET:

```
(SLET ( (variable-name initial-value) ) command* )
```

however, the semantics are different. `LET` simply names the output of a computation so that it can be referred to more than once. `SLET` sets up a state variable (ostensibly a sample-and-hold amplifier) in which values can be stored. The value of a state variable is initially set to the initial value specified in the `SLET` form, and may be subsequently changed by a `SET` command.

State variables can be used to emulate a large variety of high-level programming constructs including loops and state machines. For example, the following module uses a state variable to provide a count of the number of times that a channel makes a transition from a negative to a positive value:

```
(define-module count-transitions                              (3-9)
  (slet ((count 0)
         (last-sign 1))
    (drive output-channel count)
    (let ( (this-sign
             (signum input-channel)) )
      (if (not (= last-sign this-sign))
        (block
          (set count (+ count 1))
          (set last-sign this-sign))))))
```

3.5 Other Features

3.5.1 Syntactic Abstraction

ALFA provides a macro facility which can be used in conjunction with the SLET form to construct quite powerful abstractions. For example, a "function" which computes the integral (i.e. the running sum) of an input can be defined as follows:

```
(define-syntax (integral x)                                   (3-10)
  `(slet ( (sum 0.0) )
     (set sum (+ sum ,x))
     sum))
```

A derivative "function" can be similarly defined:

```
(define-syntax (derivative x)                                 (3-11)
  `(slet ( (last-x 0.0) )
     (let ( (temp last-x) )
       (set last-x ,x)
       (- temp ,x))))
```

The word "function" is in quotes because these forms do not define functions in the usual sense. These forms are macros, and are expanded in exactly the same way that Lisp macros are expanded. This allows the user to invoke LISP functions during the compilation of an ALFA program, and makes the compiler user-extensible.

3.5.2 Timers and Assertions

Modules can also be synchronized to real time simply by providing the value of a real-time clock through a channel. The following module outputs a square wave with a period of 20 seconds:

```
(define-module square-wave                                   (3-12)
  (slet ( (last-transition-time 0.0)
          (state 1) )
    (drive output-channel state)
    (if (> real-time-channel
           (+ last-transition-time 10.0))
      (block
        (set last-transition-time
             real-time-channel)
        (set state (- state))))))
```

This sort of real-time behavior is so useful that ALFA provides a special data type called a timer. A timer is a `BIT` (boolean) variable whose value may be set to `:TRUE` for a specified period of real time (i.e. a retriggerable monostable multivibrator), a process known as triggering the timer. Timers provide ephemeral state. (The subsumption architecture also has timers, but their periods are fixed at compile time [Brooks86], [Connell89].) For example, the previous module could have been written:

```
(define-module square-wave                                   (3-13)
  (slet ( (delay :timer)
          (state 1) )
    (drive output-channel state)
    (if (not delay)
      (block
        (trigger delay 10.0)
        (set state (- state))))))
```

Another useful ability is to be able to measure the elapsed time since an event. ALFA provides a data type called an ASSERTION which provides this functionality. The value of an assertion is the amount of time which has elapsed since the assertion was ASSERTed. The square-wave module could have been written:

```
(define-module square-wave                                       (3-14)
  (slet ( (switched :assertion)
          (state 0) )
    (drive output-channel state)
    (if (> switched 10.0)
      (block
        (assert switched)
        (set state (- state))))))
```

Note that an ALFA assertion is very different from the usual AI notion of an assertion as a statement of first-order predicate calculus inserted in a data base. ALFA assertions simply keeps track of the elapsed time since some event. They are called assertions because, by "triggering" one, the system asserts that the corresponding event has occurred. (cf. the BELIEVE construct in [Firby89]).

3.5.3 Type abstraction

ALFA provides a facility for defining new data types, allowing a primitive form of object-oriented programming. ALFA types are loosely modelled on the object system in the T dialect of Lisp. A type consists of a set of instance variables and a set of operations defined on that type. Types are defined using the following form:

```
(define-type typename
  ((var1 value) (var2 value) ... )
  expression
  ((operation . args) . method)
  ((operation . args) . method) ... )
```

The value of a state variable of type TYPENAME is the value of EXPRESSION in the environment of the type's instance variables. For example, the timer and assertion types could be defined as follows:

```
(define-type timer                                               (3-15)
  ((transition-time 0.0))
  (> transition-time current-real-time)
  ((trigger timer time)
   (set transition-time
   (+ time current-real-time))))
```

```
(define-type assertion                                  (3-16)
  ((assert-time 0.0))
  (- current-real-time assert-time)
  ((assert assertion)
   (set assert-time current-real-time)))
```

These examples assume that the current real time is available through a channel named `current-real-time`.

3.6 The Compiler

A retargetable compiler for ALFA was written which implemented all the features of the language described in this Chapter except type abstraction. Two back-ends were implemented, one for the Motorola 6811 microcontroller and another for the 68000 family of microprocessors. The 6811 is an 8-bit processor with a 16-bit address space; the 68000 is a full 32-bit processor. The 6811 is typically configured with a few thousand bytes of memory; 68000 systems regularly have several megabytes. These two processors more or less span the range of power and performance available in modern microprocessors.

In order to be easily retargetted to such a range of machines, the compiler produces P-code for a virtual reconfigurable processor. The number of available registers is given to the compiler as a parameter. The P-code also contains redundant instructions for different sorts of available addressing modes. For example, the 68000 has a stack-pointer-indirect-with-offset addressing mode which can be used to implement stack frames. The 6811 lacks such an addressing mode, and must use an index register to implement a stack frame. The P-code contains instructions for both stack-pointer-based stack frames and index-register-based stack frames. The back end simply ignores instructions for the mode that is not in use.

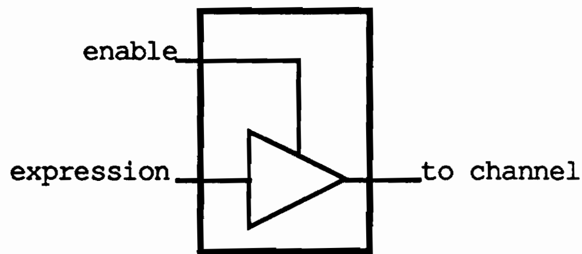
The output from the compiler is a data structure called an ALFA environment which contains not only the compiled code but also device driver links and symbolic debugging information as well. This information is used by the sequencing layer to communicate with the resulting program. Every external channel and every state variable can be accessed symbolically via the ALFA environment. However, once the program is debugged and linked, the environment can be discarded leaving only the compiled code, which is usually very compact.

3.7 Summary

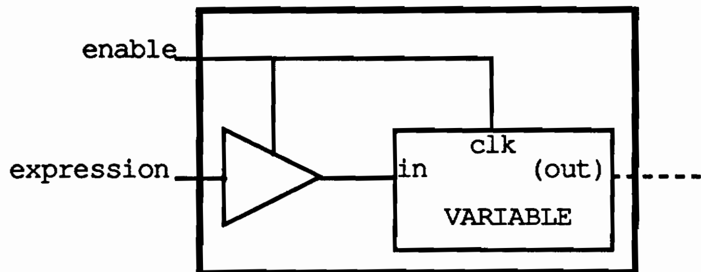
To control primitive activities we need a computational structure which computes complex (mostly) stateless transfer functions from a large number of inputs onto a large number of outputs. In order to control the complexity of designing such transfer functions we have introduced a programming language called ALFA designed for this purpose.

An ALFA program consists of a set of computational *modules* connected to each other and with the robot's sensors and actuator by means of communications *channels*. Modules compute simple transfer functions from channel outputs onto channel inputs. Channels combine large numbers of inputs onto a single output, allowing simple transfer functions to be assembled to produce more complex transfer functions.

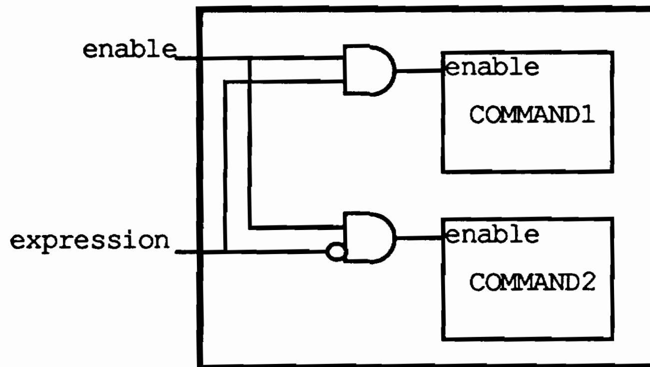
The semantics of ALFA are grounded in analog electronic circuits. (See figure 3-5.) This has two effects. First, it guarantees that any program written in the language can be compiled into a discrete simulation of an analog circuit which can complete a cycle in a constant-bounded amount of time. Second, it means that many familiar constructs of traditional programming languages are either not available or will not work as one might expect. In particular, the language contains no looping constructs, sequential assignment of state variables should be avoided, and feedback assignment must be used with care.



(DRIVE channel expression)



(SET variable expression)



(IF expression command1 command2)

Figure 3-5: Analog circuit implementations of the three fundamental ALFA constructs.

Chapter 4: Discussion and Evaluation

This chapter evaluates ALFA and ATLANTIS to see how well they meet the criteria we established for them in Chapter 1. Recall that we wanted a system for controlling autonomous mobile robots which would be robust in the face of limited computing resources, an unpredictable environment, and unreliable sensors and actuators.

Of course, the only really convincing test of the system would be to install it on a real robot and watch what happens. We will do this in Chapter 5. The purpose of this section is to go into more detail about how ALFA and ATLANTIS work in principle. This chapter also compares ALFA and ATLANTIS to previous robot control languages and architectures.

4.1 Robustness

Robustness is difficult to define. Ideally we would like our robots to always achieve their goals in a provably correct and optimal way. Since this is not possible we have to set our sights lower. Because sensors and actuators are not perfect, computation is limited, and the world is unpredictable, our robots will occasionally make wrong turns, or have to stop and think. This is probably acceptable behavior -- most of the time. People occasionally make wrong turns and stop to think, and we mostly manage to get along the world, and occasionally even do something useful. It's OK for a robot to make a wrong turn -- unless that wrong turn ends up taking the robot over the edge of a cliff. It's OK for a robot to occasionally stop and think -- unless it stops to think in the middle of a busy intersection.

A robot control architecture is robust if it makes the robot do the "right thing" a large portion of the time and the "wrong thing" a small portion of the time. "Doing the right thing" means taking actions which result in progress towards the robot's goals; doing the "wrong thing" means taking actions which actively hinder progress towards the robot's goals. (Another descriptive term for doing the wrong thing is, "making a catastrophic mistake.") In between doing the right thing and doing the wrong thing is a large class of actions which are neither right nor wrong, but just sort of there. Doing nothing is usually

in this class, except in cases where the robot's goals include meeting deadlines, in which case waiting too long can be the wrong thing.

The assessment of "rightness" and "wrongness", and thus of the robustness of the system, is highly context-dependent. Whether an action is right or wrong (or neither) depends on the robot's goals, the situation it happens to find itself in, the robot's physical limitations, any *a priori* knowledge it may have about the world, etc. Colliding with obstacles is almost always the wrong thing because this can result in hardware damage that can permanently hinder the robot in the achievement of its goals. On the other hand, if, for example, the robot occasionally needs to push objects around the room then a permanent aversion to collisions might not be the right thing.

Preventing the robot from doing the wrong thing requires doing two things: 1) detecting a situation in which wrong actions are possible (let us call these *dangerous* situations) and 2) preventing those actions from happening without restricting the robot in its choice of remaining actions. Some dangerous situations can be directly sensed given the right hardware. For example, in most indoor environments a large class of potential collisions can be detected with infrared proximity sensors. Nevertheless, there are obstacles that such sensors will not detect, such as very dark objects or non-somatic obstacles such as tangles of wire. In such cases, additional sensors, or some means other than direct sensing, must be used.

There is a class of dangerous situations which are nearly impossible to detect by direct sensing. A robot that happens to wander into the elevator by mistake when it was trying to find the broom closet can be stuck there for a long time if it has no way to push the buttons. (The situation can be especially grim if the robot leaves the elevator on the wrong floor and needs to get back on.) This situation is not unlike a person who gets on a bus or a train heading to the wrong destination. Such situations are dangerous because the robot (or the person) is stuck for a long time in a situation where it cannot achieve its goals. Even if there are no tight deadlines, the resources which are necessarily expended in the interim (battery power in the case of the robot, time and money in the case of the person) can cause the agent to fail to meet one or more of its goals.

These kinds of situations are difficult to detect because the locally available sensor information is nearly (or often exactly) identical to some non-dangerous situations. To distinguish them, in general, requires some thought and the use of stored state information.

As humans we make life easier for ourselves by modifying our environment in ways that help us to tell the dangerous situations from the non-dangerous ones (e.g. by putting up signs). We thus reduce the problem back to the first sort of situation where the required information is available locally. In cases where such modifications have not been made people commonly do the wrong thing (often with litigious results).

Detecting dangerous situations in the absence of local indications requires a certain amount of time-consuming deliberative computations. Driving to Hoboken without the package that I need to deliver is the wrong thing, and I am much more likely to forget the package if I am in a hurry than if I stop to think for a while before I leave. This gives me time to mentally project my course of action into the future to see if there are any potential future problems that require corrective action now.

Making a robot do the right thing is similar to preventing it from doing the wrong thing, but with two important differences. First, the information required to do the right thing is rarely available locally. This is because the number of potential goals an agent might have is much larger than the distinct classes of potential dangers it might face. There are millions of places in the greater Hoboken area I might want to go, but the number of dangers I have to avoid along the way to any of them is more or less the same. There are exceptions to this rule, of course. We put up signs at street corners to make it easier to extract useful information about the right thing to do next from the local environment, but the sort of information provided by street signs invariably needs to be combined with some long-term persistent state information in order to be of any use. I can determine that I am at the corner of Clinton and 12th by looking at the street sign, but in order to use this information to get to my friend's apartment I have to know that it is at the corner of Clinton and 13th, information that the sign does not provide. By contrast, to take the appropriate action in the case of a sign that says "STOP" usually does not require any state information.

The second difference is that doing the right thing is always less important - by definition - than not doing the wrong thing. If I waste some time driving randomly around the maze of one-way streets in the vicinity of 13th and Clinton for a while before arriving at my destination, my path may be sub-optimal, but it is acceptable nonetheless. Inadvertently going the wrong way down one of those one-way streets, on the other hand, is not acceptable. It is possible that, had I taken some of the computational resources that I was using to look for "one-way" signs and expended them instead on planning, I might have reached my friend's apartment sooner. However, the cost of doing this was the risk

of a collision which might have prevented me from ever reaching my friend's apartment. In cases where safety and optimality compete for resources a robot control architecture must be able to insure that safety will prevail.

ATLANTIS provides mechanisms both for preventing wrong actions and for encouraging right ones. For dangerous situations which can be detected from locally available sensor information, modules can be installed in the control layer to detect those situations and restrict the robot's actions, either by seizing control of an actuator through a prioritized channel, or by placing a limit on the value of an actuator output through maximum and minimum channels. In cases where wrong actions can be prevented by planning, ATLANTIS provides data paths from an asynchronous planner (running in the deliberative layer) to the sequencing layer which can then use this information to enable low-level activities which will not lead the robot into danger.

In cases where the robot does the wrong thing because the information was not available locally and there was not enough time to plan, the best we can do is detect the problem and try to take corrective action. Thus, ATLANTIS supports the principle of cognizant failure, that is, that wrong actions should be detected when they occur. The sequencing layer provides a sophisticated mechanism for recovering from such failures.

4.2 Limited Computational Resources

The remarkable advances in computing technology over the last twenty years have made it tempting to stop thinking of computation as a scarce resource. Unfortunately, while not limitless in any application, limitations on computation are particularly acute in mobile robot applications. The reason for this is that large computers require large power supplies[1], and large power supplies are heavy. Heavy payloads require a sturdier, and thus heavier, robot to carry them. Heavier robots require larger motors, which add still more weight and require more power. This positive feedback cycle makes the cost of computation on a mobile robot quite high.

---

[1] "Power supply" here does not mean simply a power conversion unit. The source of power must also be carried on board. Given current technologies, this means batteries or, for outdoor applications, generators powered with fossil fuels.

Computational resources fall into two classes: memory and processing cycles. Both of these have a direct energy cost. Current CMOS technology processors consume power in direct proportion to their clock frequencies. (This is because nearly all of the energy cost of running such a processor is incurred during the switching of a logic gate.) Memory consumes power both according to its speed and its size. The current state-of-the-art in onboard computation is arguably the CMU Navlab [Stentz90], which is a modified van and thus can draw on the power of an internal combustion engine. The Navlab carries a number of Sun workstations on board, and a few more can probably be shoveled in in a pinch. Nonetheless, the Navlab would be hard pressed to accommodate, say, a Cray Y/MP or a Connection Machine, and any mobile robot architecture which relied on having that much processing available would probably be of little utility.

The current state-of-the-art in onboard computation for an indoor vehicle is (again arguably) FANG[2], the JPL indoor navigation testbed, which carries a VME-based 68020 processor, a Macintosh II, and an IBM PC portable, as well as about ten 8-bit peripheral processors. This vehicle weighs several hundred pounds and just barely fits through a standard doorway. (FANG cannot go through doorways autonomously.) A more typical mobile robot is the one used in some of the experiments in this thesis (to be described in Chapter 5) which had a 68000 processor and 512K bytes of memory. Regardless of the actual numbers, computation is a scarce resource and thus must be husbanded carefully.

ATLANTIS deals with this issue by separating the critical computations from the non-critical ones, and highly optimizing the former. The control layer, which is the most critical to the functioning of the robot, is programmed in a language which compiles to very small, efficient code with very little runtime support required. (ALFA has been used to program a robot with less than 5k of memory.) The language contains no loops, and so a complete cycle of the control layer can be run in a fixed amount of time, typically less than a second in implemented applications. This allows the control layer to run at a sampling rate comparable to the mechanical bandwidth of most robot actuators with cycles to spare.

The sequencing layer is also structured in a way which makes it quite efficient. It does no search, and the caching-computation mechanism prevents needless recomputation. The deliberative layer is not optimized in any way, but the system does not depend on fast

---

[2]FANG: Futuristic Autonomous Navigation Gizmo

responses from it. Thus, it can run on slow hardware without adversely affecting the robot's safety (although it can affect the efficiency with which the robot carries out its tasks).

4.3 Unpredictable Environments

Unpredictability arises in many different ways. Some things, such as incidents of errors due to thermally induced noise, are inherently unpredictable. The rates at which such errors occur can be predicted, but the times at which they will occur can not be, even in principle.

Some things cannot be predicted because the computations required to compute them cannot be performed, either because no decision procedure exists (e.g. the halting problem) or because the decision procedure is computationally intractable (e.g. the travelling salesman problem).

Some things cannot be predicted because the information required to predict them is not available. For example, I live in a residential neighborhood where most people park on the street. The street has a lot of through-traffic, but the configuration of parked cars tends to change fairly slowly. Nevertheless, I cannot predict the location of an empty parking space when I come home from work until I round the corner and look to see where the empty spaces are. Once I see an empty space with no cars nearby, I can predict with a high degree of accuracy that it will stay empty until I park in it.

Predictions must be made with great care because a wrong prediction can be disastrous. If I base my actions exclusively on a prediction that a particular parking space will remain empty for a while, then I will wreck my car if that prediction turns out to be wrong. On the other hand, if I never predict anything then I am paralyzed. I cannot, for example, find my car without predicting that its location will remain unchanged from where it was last parked. To use predictions properly we must be careful to do two things. First, we must attempt to predict only aspects of the world which are predictable. Second, we must use our predictions to guide our actions but not to control them.

The use of predictions and plans is intimately related to the issue of the use of stored state information. When a robot stores a piece of internal state it makes an implied prediction that the information contained in that state will still be valid some time hence.

The difficulties encountered through the careless use of internal state have caused some researchers to advocate abandoning internal state entirely, or at least abandoning persistent internal state [Connell89]. This is tantamount to claiming that nothing in the world can be usefully predicted, or that nothing can be predicted for an indefinite period of time. While the latter claim may be true in the limit, there are aspects of the world which can be reliably predicted over extremely long periods of time. As we observed in Chapter 1, such predictions can typically be made only at high levels of abstraction.

Nevertheless, even though many things can be accurately predicted, nearly nothing can be predicted with absolute certainty. Thus, predictions should be used to *guide* actions and not to control them. In ATLANTIS, the predictions and plans made by the deliberative layer do not control the robot directly, but rather provide input to the sequencing and control layer which actually control the robot. The deliberative layer can suggest going over to the parking space over there, but if a car pulls into it while the robot is *en route* the control layer will (presumably) prevent a collision.

4.4 Unreliable Sensors and Actuators

Noise is the deviation of the behavior of a physical system from a deterministic mathematical model of that system. This definition is relative to a model. Thus, one aspect of the behavior of a system may be noise under one model but not under another. The distortion of an audio amplifier is noise under a linear model but not under a non-linear model. Thermal noise on the other hand, because of its quantum-mechanical source, is noise under *any* deterministic model. All sensors are noisy to some degree. (See [Everett89] for a good review of the current state of the art in real-world sensors for mobile robots.)

Quantum mechanical and thermodynamic noise are pervasive and inescapable. A vast amount of engineering goes into reducing this kind of noise to the point where it can be safely ignored. The modern digital computer is arguably the best example of engineering away noise. We purify silicon and ferrous materials so that we can make transistors and magnetic media with very few defects. We build parity-checking memory. We design error-correcting codes. The result is usually a system which behaves like its mathematical idealization with an extremely high degree of fidelity. Nevertheless, glitches do happen, though they are exceedingly rare. We can even calculate with a fair degree of accuracy the

rate at which noise in the system will cause a bit to be inverted. What we cannot predict is when this will happen, nor can we ever make the rate fall to zero.

As challenging a problem as engineering away noise problems in computers is, doing the same for mobile robots is vastly more difficult. The reason for this is that when we build a computer we construct a complete environment for the device to operate in. We do not allow arbitrary configurations of doped silicon on our chips, but only a very few, very stylized configurations that we can understand, in repetition upon repetition. Our chips are connected using printed circuit traces whose geometry is carefully controlled. We put the resulting boards in climate-controlled boxes and allow them to interact with the world only via precisely characterized electrical signals at carefully measured frequencies. We can engineer away the noise in a computer only because we can engineer the entire system.

The same approach cannot work for autonomous mobile robots because one of the ground rules is that we cannot engineer the environment in which the robot is to exist, at least not to such an extent. The object of the game is to build a machine that can operate in an unstructured, unengineered environment. Without this restriction, building an autonomous mobile robot is not an interesting problem. Autonomous mail-delivery robots which follow a painted line on the floor can be purchased off the shelf. The challenge is to build a robot which will work without the painted line so that we can send it to Hoboken where there are lots of lines (except where the road construction has ripped up the pavement), or to Mars where there are no lines, or through our living room, where we prefer to leave the shag carpeting unstriped, where we sometimes leave the laundry strewn about, and where the family dog occasionally wanders through. Engineering away all noise in such unstructured conditions is a Quixotic endeavor.

We therefore must deal with the reality that sensory input and actuator output will occasionally deviate from our models, often by significant amounts. This is not a particularly difficult problem, but it is one that the research community has been seemingly reluctant to come to grips with. Many papers are published on "provably correct" or "optimal" path planning strategies which assume noiseless sensors and actuators (e.g. [Shiller90]). Optimizing a path down to the last centimeter or the last second is of questionable utility, especially when the locations of obstacles are only known to the nearest half meter, or the optimization process itself takes many minutes. (But see [Dean90a] for a good example of a formal approach to dealing with sensory noise.)

Filtering out noise involves matching the actual input data against a partial expectation of what that data should look like in order to deduce what is noise and what is not. The simplest example of a filter is one which simply discards all readings outside of a range of "reasonable expected" values. More complex filters can match data to more complex probability distributions or to descriptions of the expected frequency content of the signal [Ogata87].

The need to have an *a priori* description of the signal might seem like a drawback, but a brief reflection will reveal that there is no way around it. Without any expectations about what a signal should look like there is simply no way to tell whether it is noisy or not. For example, consider the following two noisy data streams:

TR#NSMISSION ERxORS CAN BE AjROIDED B@ PRITY CHECXING
B0016724394002182047FF08E5482C30000008060000

It is easy to tell that there are errors in the first stream but not in the second. The reason is that the first resembles English text strongly enough that you bring to bear strong expectations about what such data is supposed to look like. These expectations are so restrictive, in fact, that you were probably able to not only detect the errors, but also to correct them. The second data stream, on the other hand, appears at first glance to be just a string of random hexadecimal digits for which the restrictions which you bring to bear are far less restrictive. It is only if you know that the stream represents an instruction sequence from a machine language program for a 68000 microprocessor[3] that you can tell where the error is.

The usual way to do the math required to filter out noise is to just do the math, i.e. to perform the required computations as deliberative computations. However, there are many cases where filtering can be performed implicitly by an agent interacting with an environment. For example, consider a robot which engages in a low-level activity which makes it go through a door while staying as far as possible from obstacles (e.g. door jams). Upon successful completion of this activity, the robot will end up very close to the

---

[3]In fact, it is an instruction sequence from Microsoft Word version 4.0, the word processor on which this thesis was written.

center of the door, assuming that the bandwidth of the robot's range sensors is high compared to the speed of the robot. This is because, although any particular range-sensor reading may be wrong, no one reading governs the actions of the robot for very long. Thus, the readings are averaged, not by any computation, but by the low-pass filtering performed by the robot's actuators. Chapter 5 describes a real robot which fixes its position in exactly this way.

ATLANTIS naturally supports this sort of implicit filtering, as well as the traditional deliberative method. Implicit filtering is simply an emergent property of a large class of primitive activity control functions. Cognizant failure is the mechanism by which the control layer notifies the sequencing layer that the amount of noise encountered was too great to be handled implicitly, and that something else needs to be done. Explicit filtering, where it is necessary, is performed, like all other time-consuming computations, by the deliberative layer. The next chapter describes real robots which use both techniques for dealing with noise.

4.5 Related Work

ALFA and ATLANTIS[4] bear various degrees of resemblance to several previously proposed languages and architectures for robot control. It is therefore worthwhile to compare and contrast the present work with previous efforts.

4.5.1 Subsumption

There are really two subsumption architectures, and this has been the root of a great deal of confusion. The subsumption architecture was first introduced by Brooks in [Brooks86] and subsequently updated in [Brooks89]. A modified version was developed by Connell [Connell89].

---

[4]ATLANTIS: The Layers Are Neater Than In Subsumption

4.5.1.1 Brooks' Architecture

Brooks' subsumption architecture is more a design methodology than an architecture. Rather than specifying a set of components and the interfaces between them, subsumption specifies a set of guidelines to be used for developing control mechanisms. To quote Mataric, "Rather than a recipe for programming robots, [the subsumption architecture] is a set of philosophical concepts about robot ... design." [Mataric90]

Brooks' original formulation of the architecture revolved around the idea of decomposing the problem of robot control by task rather than by function. Most robot control architectures are composed of functional modules which do such things as sensor processing, planning, execution monitoring, etc. Brooks argues that such a design is inherently inefficient because it forces each functional module to be powerful enough to support any task the robot may be called upon to perform.

Rather than develop general functional modules, the subsumption architecture advocates the development of more narrowly focused mechanisms called behaviors. Each behavior is designed to control only a single task, allowing the computation within the behavior to be optimized for that task. Each behavior is coupled directly to the robot's sensors and actuators. Conflicts among behaviors are resolved by an arbitration mechanism whose structure is not specified by the architecture.

There are some other general guidelines advocated by the architecture. Each individual computational module should be fairly simple. To date, implementations of the architecture have structured computational modules as simple finite-state machines. More recent implementations have provided abstraction mechanisms for organizing large numbers of FSA's into coherent groups [Brooks90], but the underlying computations have remained very simple. In particular, the architecture strongly opposes the use of centralized data structures which can be accessed across behaviors. Each behavior is responsible for maintaining whatever data structures it needs. The only way information is shared among behaviors is by means of messages sent over a low-bandwidth, unreliable communications network. There is no centralized control and no hierarchical decomposition. Every behavior operates at the same low level of abstraction.

4.5.1.2 Connell's Architecture

Connell's version of the subsumption architecture is a more restrictive version of Brooks'. Rather than using finite-state machines as the computational elements, Connell uses a mechanism which is similar to a simple ALFA module. A computational element of Connell's architecture consists of a transfer function and an applicability predicate. The module outputs the value of the transfer function whenever the applicability predicate is satisfied. Connell also allows a single bit of ephemeral state to keep the output of the module enabled for some time after the applicability predicate is no longer true. In other words, a module in Connell's architecture is nearly equivalent to one of the following two pieces of ALFA code:

```
(IF applicability-predicate                                    (4-1)
  (DRIVE OUTPUT transfer-function))

(SLET ( (MODE :TIMER) )                                        (4-2)
  (COND
    (initiation-clause (TRIGGER MODE some-constant))
    (satisfaction-clause (TRIGGER MODE 0.0)))
  (IF MODE
    (DRIVE OUTPUT transfer-function)))
```

There are two main differences between a Connell-style module and the above ALFA code. First, Connell's system is built on a substrate of finite-state machines and discrete-message communications. Connell uses this substrate to emulate continuous communications, but the discrete nature of the substrate is not hidden, resulting in cumbersome code. Second, connections among modules are made by means of *wires*. Wires were introduced as a software construct by Brooks. Connell's implementation is notable in that it used a physically distributed network of processors connected together by physical wires. Wires are somewhat like ALFA channels but with three major differences: 1) a wire has only a single input, 2) messages in one wire can suppress messages in another wire, and 3) the connections between wires and modules are specified in the definition of the wire rather than the definition of the module. This makes sense when one is dealing with physical wires, but as a software construct it leads to a number of problems.

Perhaps the most serious problem with wires is that it is never possible to know whether previously developed computations will still work properly when new modules are added, since one can never know when some critical signal will be suppressed [Rosenblatt89]. In ALFA, new modules cannot interfere with the internal computations of

existing modules. Furthermore, since priorities (when they are used) are set within the definition of a channel, it is possible to guarantee that certain critical pathways cannot be suppressed unless the channel is redefined. In other words, it is not possible to interfere with a working computation in ALFA simply by adding code.

4.5.1.3 Critique

The subsumption architecture has been criticized on the grounds that it will not scale to complex tasks.[5] This criticism has been consistently answered by example after example of physical robots performing ever more complex tasks. Connell's architecture is so restrictive that one might wonder if it were capable of much of anything, yet it was successfully used to control one of the most sophisticated examples of an autonomous robot to date. Connell's robot was able to navigate in an unknown environment, locate a soda can by means of an active vision system, collect the soda can in a gripper, and return to its starting point. Recently, Mataric has demonstrated a robot based on Brooks' less restrictive architecture that was able to construct a map of its environment and plan paths using that map [Mataric90]. The map and the planner were implemented in a distributed fashion according to the tenets of the architecture. Maes has used the architecture to produce an impressive example of reinforcement-based learning on a physical robot [Maes90]. Horswill has used the architecture to control a robot chasing a moving target [Horswill88].

Most of these tasks share a common feature: they do not require the maintenance of large amounts of state information. (Mataric's work is a notable exception.) The range of tasks which can be accomplished without internal state is large, much larger than was once supposed, but there is a limit. For example, Connell's robot sometimes took very circuitous routes because it lacked a map of the world, and there were some places its navigation scheme couldn't reach at all [Connell89].

Mataric's work addresses this issue. Her system builds up an incremental topological map of the world based on sensory features, and then uses that map to plan paths to goals. An interesting feature of this work is that the high-level system (the map and planner)

---

[5]Rodney Brooks, personal communication.

interfaced directly to the robot's actuators. However, her robot operated in a fairly simple domain (essentially a one-dimensional world) and it is not clear that her method will extend to more complex tasks. If it does, it will not be the first time that subsumption has answered its critics with experimental evidence.

However, even if subsumption is extended to cover such cases, it will have to address the issues raised in this thesis, namely, that the calculations required to manage large amounts of state information are often inherently time-consuming, even when parallelized. Thus, the state information must be stored and processed at a high level of abstraction, and the results cannot be used to control the robot directly. In Mataric's work, the state information does exist at a high level of abstraction (as a topological map) but the results of the planning process do control the robot's actuators directly. This worked in Mataric's case because she used a synchrodrive robot which could be stopped and rotated in place by the planning layer without fear of colliding with obstacles. In general, for a high level system to usurp control of the robot's actuators without risking collisions, it must completely subsume the functionality of the lower control levels.

4.5.2 RAPs

Firby's Reactive Action Package (RAP) system [Firby89] has already been discussed in Chapter 2. The sequencing layer in ATLANTIS is essentially a stripped-down RAP interpreter modified to control activities which might overlap. To accommodate this change, a system of semaphores and an unwind-protect mechanism had to be added to the original RAP formalism. Also, much of the machinery of the original RAP system involved the identification of objects and the coordination of multiple conflicting tasks, abilities which were largely unnecessary for the navigation tasks investigated in this work. However, the fundamental structure of the ATLANTIS sequencer is strongly based on RAPs. The idea of cognizant failure has its roots in the RAP work. ([Noreils90] also deals with detecting and recovering from failures.)

The RAP system consists of two main parts, the RAP library and the RAP interpreter. The RAP library is simply a collection of RAPS which are themselves collections of methods for accomplishing something. Each method is annotated with information that describes under what circumstances it is applicable. RAPs are written in a special language designed for this purpose. The RAP interpreter is a program which executes the

procedures described in this language. The interpreter runs in cycles. The system starts with a set of tasks stored in a data structure called a task queue (which, incidentally, is not a queue). At the beginning of each cycle the interpreter chooses a task from the queue based on a set of heuristics. It then finds a RAP in the RAP library for performing that task, and chooses one the the RAP's methods based on the method annotations and the perceived current state of the world. A method is either a primitive (discrete) action, in which case it is executed, or it is a set of tasks, in which case these are placed on the task queue. The cycle then begins again. Each RAP keeps track of whether or not it has succeeded in accomplishing its goal, and keeps trying methods until either it succeeds, or all applicable methods have been tried several times. Because a task has control of the queue for only one cycle, the system can respond quickly to unexpected occurrences in the world.

The RAP system was designed to be the middle layer of a three-level architecture [Hanks90a]. In the original RAP architecture, the bottom layer controlled discrete-operator-style actions, and the top layer was simply[6] a planner which generated RAPs. In ATLANTIS the top layer has been generalized to perform all manner of time-consuming computations, including sensor processing as well as planning. Furthermore, ATLANTIS has an explicit feedback data path between the sequencing layer and the deliberative layer which the RAP system did not have. The RAP planner simply planned away and installed RAPs into the RAP library. In ATLANTIS, the computations are directed by the sequencing layer, so computational resources are allocated to the task at hand.

The RAP architecture also relied upon a few unrealistic assumptions which have been discharged to some degree in ATLANTIS. First, RAPs assumed that the RAP interpreter consumed no time. Second, RAPs assumed that all sensor information was installed by the lower level into a data base which was the sole repository of information about the world.

In ATLANTIS the time consumed by computations is taken into account. ATLANTIS structures computations in such a way that the time-critical ones happen in a timely way. The control layer, which is the most time-critical component of the system, is programmed using ALFA which guarantees an upper bound on the time required for a computation to

---

[6]"Simply" in this sentence means that the planner was *only* a planner, and did not perform other deliberative computations such as sensor processing. It does not mean to imply that this planner is simple. In fact, constructing a planner to generate RAPs is an open research area.

complete. The sequencing layer, whose performance affects efficiency more than safety, uses cachable functions in order to remove redundant computations and thus speed up the system wherever possible. All computations which are truly time-consuming are banished to the deliberative layer where they run asynchronously.

Sensor information in ATLANTIS is also handled quite differently. This is largely due to the difference in the sorts of sensor information used by the two systems. The sorts of tasks which the RAP system dealt with involved remembering the locations and properties of objects in the world, an ability which is largely irrelevant when simply moving from place to place. It is interesting to note that navigation, which was largely ignored by RAPs, required handling sensor information differently, but still fits very neatly into the RAP execution model.

A second-generation RAP system is under development by McDermott [McDermott90]. The system includes a transformational planner called XFRM and a plan description language called RPL (Reactive Plan Language). Like RAPs, XFRM and RPL currently ignore low-level control.

4.5.3 TCA

Simmons' Task Control Architecture (TCA) [Simmons90] is also very similar to ATLANTIS and RAPs. TCA consists of a set of task-specific computational processes called *modules* which communicate with each other by passing messages through a *central control module*. The central control module routes messages dynamically among the task modules. Tasks in TCA are structured as hierarchical *task trees* which encode parent-child relationships among messages. A task tree is similar to an expanded RAP. TCA allows concurrent execution of steps in the task tree, which can include computational as well as physical tasks. TCA includes mechanisms for enforcing temporal constraints among the various steps in the task tree.

The principle difference between TCA and ATLANTIS is that TCA is designed top-down while ATLANTIS is designed bottom-up. The methodology of TCA is to "first develop systems having sequential sense-plan-act cycles, then use the TCA facilities to add concurrency." Monitoring and error handling are also added after the code for handling "normal" situations is in place [Simmons90]. TCA does not specify the structure of the low-level control mechanisms at all.

In ATLANTIS the development cycle is exactly the opposite. Primitive activities are engineered first. Error detection and recovery are incorporated into the design at every stage rather than added afterwards. Deliberative computations are only used when there is not enough information available locally to make a correct decision.

Which of these approaches is preferable is, of course, an empirical question. To date, TCA has been used to control two mobile robots, a small indoor robot which collects cups, and the CMU Ambler, a large, six-legged walking robot. It should be noted that the cup-collecting robot used an overhead camera to provide the robot a global view of its environment. All of the robots controlled by ATLANTIS to date have used only sensors mounted on the robot itself.

4.5.4 AURA

Arkin's AURA architecture [Arkin90] is superficially very similar to ATLANTIS. It, too, is an architecture intended to control autonomous mobile robots in the real world. However, AURA is motivated by different goals, and uses very different approaches to solving many of the problems addressed by both systems.

AURA is motivated a great deal by neurophysiological evidence, whereas ATLANTIS is motivated more by engineering considerations. AURA's fundamental building block is a motor schema, which is a vector field associated either with a goal or with an obstacle. Motor schemas are combined by vector addition, resulting in an overall vector field which control's the robot's motion. A planner modulates the motor schemas to keep the robot out of local minima which are often produced when combining such vector fields [Slack90].

Much of the effort in AURA has been devoted to vision processing and building up an accurate world model, an issue which is largely peripheral to ATLANTIS. Because ATLANTIS does not commit itself to a vector-field model of local navigation it does not need a world model. On the other hand, ATLANTIS can make use of such a model if it is available. (In the next chapter we will see examples of robots which navigate to a goal both with and without a world model.) Another difference between ATLANTIS and AURA is the way in which world models are used. In AURA there is a direct connection between the global world model and the low-level motor schemas. In ATLANTIS, any information from a global world model maintained by the deliberative layer which is used by the control layer must get there by means of the sequencing layer. This is important because global

world models tend to be dynamic data structures which can grow quite large, and thus require a long time to process. The sequencing layer must "filter out" a fixed amount of relevant information to pass to the control layer. This way, response time can never be compromised by the time required to process a large world model. To a large extent these differences are a reflection of the differences in motivation between the two architectures.

4.5.5 Situated Automata

Rosenschein and Kaelbling's situated automata theory [Kaelbling90] is a design for a robot control architecture with sound theoretical foundations. The basic architecture consists of two components, a perception component and an action component. The perception component consists of a network of combinatorial logic gates connected to the robot's sensors as well as to its own outputs through a time-delayed feedback loop. The action component is simply a combinatorial logic array. The intuition is that the perception component keeps track of the perceived "current state of the world" while the action component maps that perception onto an appropriate action for achieving the robot's goals in that situation. The feedback loop in the perception component allows the robot to remember the past, and thus allow past data to be incorporated into current decisions.

The architecture is supported by an elaborate theory of the semantic information content of various components of the network [Rosenschein86]. Kaelbling has also developed two development tools (GAPPS and RULER) for automatically generating the combinatorial logic networks from high level descriptions of the robot's goals and of the world [Kaelbling88]. There is a third tool, REX, for generating automata from low-level descriptions.

The vision of its designers is that the situated automata approach will allow high-level descriptions of environments and tasks to be compiled automatically into a reactive control mechanism about which one can make formal claims. ATLANTIS, by contrast, assumes that each layer will be designed by hand. ATLANTIS provides tools and techniques to make this process easier, but a skilled human designer is required.

Automatic synthesis of control mechanisms from high level descriptions is a laudable goal. However, the situated automata approach has a number of problems when put into practice. The implementation of the theory takes traditional planning algorithms and compiles them down to circuits which perform the computation in constant-time bounded

steps. (This is essentially the same as encapsulating a computation in an engine [Dybvig89].) At each step, the robot performs actions based on its current sensor data and the output of the planning computation, which may or may not be available. (In this, the interaction is identical to the interaction of the ATLANTIS deliberative layer and the rest of the system.) The situated automata compiler allows some of the computation to be performed at compile time, resulting in a faster, but much larger, circuit.

The situated automata architecture, like subsumption, is an essentially homogeneous approach to robot control. Although there are tools which allow the mechanisms to be described at different levels of abstraction, the same underlying computational structure is used throughout. The main problem with the approach is that it does not address the problem of allocating scarce computational resources. The circuits produced by the compilers advance all of their computations by a fixed amount at every cycle. While the system does guarantee a constant-time response, this constant can get quite large when, for example, a sophisticated planner is embedded into the circuit. This is especially true when the circuit is being simulated on a serial processor. ATLANTIS, on the other hand, only advances time-critical computations at every cycle (i.e. those in the control layer), and allows the remaining processor time to be allocated to different computations depending on the situation.

4.5.5 Payton and Rosenblatt's Architectures

Many of the features in ATLANTIS were anticipated by the architecture described in [Payton86]. Payton's architecture consists of four layers rather than three, but the general approach is strikingly similar. The top three layers comprise a traditional hierarchical planning system responsible for local path planning, global path planning, and mission planning or task-level planning. The bottom layer, however, consists of a set of reflexive behaviors which are activated by the local planner. Each reflexive behavior consists of one or more virtual sensors connected to a computational module.

Most of the interesting features of the architecture are in the design of this bottom level. Like ATLANTIS, the bottom layer of Payton's architecture is structured as a set of modules (which Payton calls behaviors) which can be activated in various combinations by the local path planner to control activities. Behaviors receive their inputs from one or more virtual sensors, which are essentially partial world models. However, unlike ATLANTIS,

Payton's architecture places no restriction on the sorts of computations which can be performed within a behavior (or, for that matter, within a virtual sensor). Command arbitration is performed by a monolithic arbitrator, which also performs unrestricted computations. Furthermore, there is a global blackboard which behaviors can use to communicate with each other. Messages written to the blackboard persist until they are explicitly erased, resulting in a large quantity of persistent state. This often resulted in hard-to-find bugs.[7]

More recently, Payton and Rosenblatt's architecture [Rosenblatt89], [Payton90a], [Payton90b] is a direct descendent of Payton's 1986 architecture and is designed to solve many of these problems. Rosenblatt's architecture abandons the global blackboard and monolithic command mediation scheme in favor of constraint-propagation channels. Rosenblatt's architecture is layered but homogeneous. Upper layers provide guidance to lower layers, but all layers have the same computational structure.

4.5.6 Pengi and Sonja

Agre and Chapman have been the most vociferous advocates in the AI community of the idea that a plan might be something other than a step-by-step procedure to be blindly followed [Agre90], [Chapman89]. They were also the first to claim that any theory of robot control must have as its foundation a theory of activity. The hierarchy of activities upon which ATLANTIS borrows heavily from their theory of everyday activity. ATLANTIS augments Agre and Chapman's theory with an account of deliberative computations.

Agre and Chapman have produced two implemented systems, Pengi and Sonja, based on their theory. Both systems operate in video-game domains. It is not really fair to compare ATLANTIS with Pengi and Sonja because they are motivated by quite different concerns. ATLANTIS is an architecture for engineering control systems for real robots, whereas Pengi and Sonja are tools for understanding and modelling human activity.[8]

---

[7]David Payton, personal communication.

[8]Phil Agre, personal communication.

While the design of ATLANTIS was guided by observations of how humans operate, producing an accurate cognitive model was not a primary concern.

4.5.7 PRS

Georgeff's Procedural Reasoning System (PRS) [Georgeff87], [Ingrand90] is an architecture for managing deliberative computations in a reactive framework. PRS is similar to RAPS, TCA, and the top two layers of ATLANTIS. PRS is notable in that it combines the scheduling of deliberative computations and the computations themselves into a unified framework. All computations are described in structures known as Knowledge Areas or KA's. The scheduling of computations is controlled by meta-KA's. PRS was used to control a real robot performing a navigation task in an indoor environment.

The main difference between PRS and ATLANTIS as that PRS has no control layer. Thus, the performance of the robot was not very robust[9] because it had no low-level cognizant failure, even though the mechanisms for dealing with such failures once they are detected do exist within PRS.

4.5.8 Other Architectures

There is a vast array of other robot control architectures in the literature. Nearly all of them are variants on the traditional sense-plan-act architecture where a planner constructs a plan from a world model to be executed by an execution system. There are innumerable variations on this theme. The most common is some sort of hierarchical generalization of the basic approach, where one planner generates a plan at a high level of abstraction which gets fed to another planner which fills in the details (e.g. NASREM [Smith89]).

Another variation is to allow some of the steps in the process to occur in parallel. The CODGER architecture [Stentz90] is a good recent example of this approach. It attempts to circumvent the inherent slowness of the sense-plan-act model by constructing incremental plans and pipelining the process. The results have been fairly impressive, largely due to the sophisticated planning algorithms used. However, like all purely deliberative

---

[9]Marcel Schoppers, personal communication.

architectures, CODGER is fundamentally limited by the speed of the pipeline. To quote Stentz, "[CODGER] is not a real-time system. Due to the relatively long latencies in TCP/IP message passing and the varying execution patterns of processes in a UNIX time-sharing environment, data transfer cannot be guaranteed within given time bounds. This limitation is acceptable given the types of navigation scenarios we have addressed. For systems with real-time constraints, however, the lack of fast, direct communications channels makes CODGER inappropriate." [Stentz90]

Another variation has been to augment the classical approach with an *execution monitoring* system which monitors the execution of the plan and takes corrective action when things go wrong (e.g. [Broverman87]). Most execution monitoring systems simply check the values of the robot's sensors as it goes along to make sure that they fall within expected bounds. When they do not, the execution monitoring system diagnoses the problem and takes corrective action. The problem is that diagnosing the problem and taking corrective action is a very hard thing to do, in general subsuming the entire problem of deciding what to do to begin with. Some systems try to get around this by anticipating possible failures and caching the required responses (e.g. [Miller89], [Gat90]).

In the extreme case, such precomputation implies generating a *universal plan* which says what to do in every conceivable situation, or at least in a large class of situations [Schoppers87]. The idea is that a universal plan can solve the problem of deciding what to do next when something goes wrong. There are a number of problems with this approach. First, generating the universal plan is at least as time consuming as generating a non-universal plan, and usually more so. Second, universal planners require that all possible states of the world be enumerable at plan-time. In the case of blocksworld domains, this means that all the blocks must be known ahead of time. In the case of route planning, it means that a global map must be known at plan-time. If a new block or a new obstacle is encountered, the system must replan from scratch.

Universal plans have some particular problems when applied to mobile robot navigation. In this case, the universal plan usually takes the form of a vector field indicating the preferred direction of travel at every point. These can actually be generated without too much difficulty. However, knowing the preferred direction of travel is not at all the same as knowing what to do, especially for robots with non-holonomic constraints. This will be discussed further in the next chapter.

Theo-Agent [Mitchell90], like situated automata, attempts to compile the results of deliberative planning into a reactive controller. Unlike situated automata, Theo-Agent uses the results of its own experience in order to guide the planning, which is done at run-time. Theo-Agent has a number of noteworthy features, among them a dependency network for updating the information derived from sensor data. Unfortunately, like situated automata, the computational overhead required to maintain the system can be overwhelming. It is not yet clear whether the system will scale to work on realistic problems.

There are a number of other notable architectures in the literature. The architecture presented in [Noreils90] addresses the problem of recovering from task-level failures. Robo-Soar [Laird91] adapts the SOAR production system architecture to robotic applications. Soldo argues that reactive and preplanned control can be usefully combined, and presents an architecture based on a construct called a behavior expert which was used to control a real robot in an indoor environment [Soldo90]. Durfee presents a similar argument and a system for controlling a robot implemented as a blackboard [Durfee90]. Andress and Kak describe an architecture for navigation and constructing composite world models from vision data [Andress88]. Anderson and Donath present an architecture based on observations of animal behavior [Anderson90]. Hammond presents an architecture which is designed to be able to take advantage of unexpected opportunities [Hammond90]. Most of these systems are homogeneous architectures. An interesting example of a heterogeneous system is presented in [Connell90] where a human is used as a high-level controller for a low-level system based on the subsumption architecture.

Finally, though not a robot control architecture, the mechanisms described by Braitenberg [Braitenberg84] were the source of much inspiration in the design of ALFA and ATLANTIS.

4.6 Summary

This chapter argued informally that ALFA and ATLANTIS would achieve their design goal of producing robust behavior in the face of noise, unpredictability and limited computation. Formal arguments are hard to obtain because of the difficulty in giving a precise definition of "robust". Robustness was defined informally as a tendency to take actions that will result in the achievement of goals and a tendency not to take actions that

actively hinder the achievement of goals. In other words, a system is robust if it usually does the right thing while seldom doing the wrong thing.

If one must choose, not doing the wrong thing is more important than doing the right thing. This is because doing the wrong thing can keep an agent from achieving its goals permanently, while not doing the right thing can be more easily recovered from. Avoiding actions which lead to catastrophic mistakes requires that the robot be able to detect "dangerous situations" in which such actions are possible. Usually, such situations can be detected directly by the robot's sensors, in which case mechanisms are provided in the control layer to turn such perceptions into appropriate constraints on the robot's actions. In cases where such actions cannot be detected directly, the deliberative layer can provide information to the sequencing layer which will allow it to avoid catastrophic mistakes. Such computations are time-consuming and may not finish before it is too late. In this case, the best we can do is to detect the problem after it has occurred and try to recover from it. The sequencing layer supports this functionality.

Noise is handled in two ways: by performing deliberative filtering computations (the traditional way), or implicitly by monitoring the performance of primitive activities. The mechanism of cognitive failure allows the system to recover in cases where the noise is too great to be handled implicitly.

Unpredictability is handled by attempting to predict only those aspects of the environment which are predictable. The use of internal state is intimately related to making predictions; state information contains an implicit prediction that the information stored therein will still be valid some time in the future. Thus, state information must be matched with the predictable aspects of the environment. The world tends to be predictable only at high levels of abstraction, in which case it can often be predicted with a high degree of accuracy. Finally, predictions should be used to guide actions but not to control them. All of these principles are embodied in ATLANTIS. Most of the state information in the system is maintained by the deliberative layer at a high level of abstraction. It is used to guide the robot's actions by providing advice to the sequencing layer, which enables modules in the control layer, which controls the robot.

Broadly speaking, the main difference between ALFA and ATLANTIS and other robot control systems is the heterogeneous nature of ATLANTIS, and the resulting restrictions on ALFA. Nearly all previous attempts to combine reaction and deliberation have done so

by embedding one within a computational framework designed for the other. A number of special-purpose languages have been designed to try to combine reaction and planning. ALFA is explicitly designed not to do everything. It has severe restrictions placed on it specifically designed to make deliberative computations inconvenient.[10] This allows the language to specifically address those issues which are unique to reactive control. Designing the language to interface to a deliberative system implemented using existing technology allows the overall system to be just as powerful as a homogeneous system, but without the added complexity.

---

[10]It is theoretically possible to do arbitrary computations in ALFA because the language is Turing-complete.

Chapter 5: Experiments

The preceding Chapters described ATLANTIS[1], an architecture for controlling autonomous mobile robots. This Chapter will attempt to demonstrate that ATLANTIS works as advertised, that is, that it produces robust behavior in unstructured, unpredictable environments in the face of noisy sensor data and limited computational resources.

Ideally one would like to prove this claim. Unfortunately, it is very difficult to prove anything about ATLANTIS because it is hard to give rigorous definitions for terms like "robust behavior" and "unstructured environment". To quote Firby, "Without rigorous definitions to prove things about, evaluation of the system must lie in actual performance." [Firby89]

To support the claims about the performance of ATLANTIS, an extensive series of experiments was conducted on three real robots and on a simulator. These experiments are described in this Chapter.

Guide to Chapter 5

This Chapter is quite long. To make reading it easier, the following is a brief guide to its contents.

Chapter five describes four major experiments.

Section 5.1 describes Tooth, a small robot with minimal computational resources. This robot was programming in ALFA to collect small objects and deposit them near a beacon.

Section 5.2 describes Toto, an indoor robot developed at MIT, which performed a complex navigation task under the control of an ALFA program guided by a rudimentary sequencer.

Section 5.3 describes Robbie, the JPL planetary rover testbed, which was used to demonstrate the coordination of multiple physical and computational activities in real time. This robot performed a complex navigation task and demonstrated the ability of

[1] Architectures That Lack ATLANTIS' Nifty Techniques Inhabit Simulators

ATLANTIS to strategically replan in real time at a high level of abstraction based on information acquired at runtime.

Section 5.4 describes a series of experiments performed on a real-time simulated version of Robbie. These experiments demonstrate the ATLANTIS architecture controlling a robot performing multiple tasks in a complex environment.

5.1 Tooth

This section describes an experiment using ALFA by itself to control a small indoor robot performing an object-collection task. This experiment demonstrates the abilities of ALFA to control complex primitive activities. It also demonstrates some rudimentary sequencing capabilities constructed directly in ALFA.

5.1.1 The Robot

Tooth[2] [Angle89] is a small indoor robot, approximately 30 cm long, 20 cm wide. (See figure 5.1-1.) It has front-wheel steering, two independent rear-wheel drive motors, and a collection of contact sensors and infrared proximity sensors which are used for obstacle avoidance. It also has a two-degree-of-freedom gripper with which it can pick up small objects. The robot has five actuators: two drive motors, the steering motor, and two motors to control the gripper, one to open and close it, and the other to raise and lower it. There are ten one-bit sensors: five bump sensors, two grasp sensors, two proximity sensors, and one infrared break-beam sensor for detecting when an object is in the gripper. There are also ten analog sensors: eight cadmium-sulfide photo cells (CdS cells) and a tachometer on each drive motor. The robot's computing resources consist of two Motorola 6811 microcontrollers, each with 2k bytes of electrically erasable programmable read-only memory and 256 bytes of random-access read-write memory.

---

[2]The name is a result of the robot's lineage. Tooth is the smaller brother of FANG, the Futuristic Autonomous Navigation Gizmo.

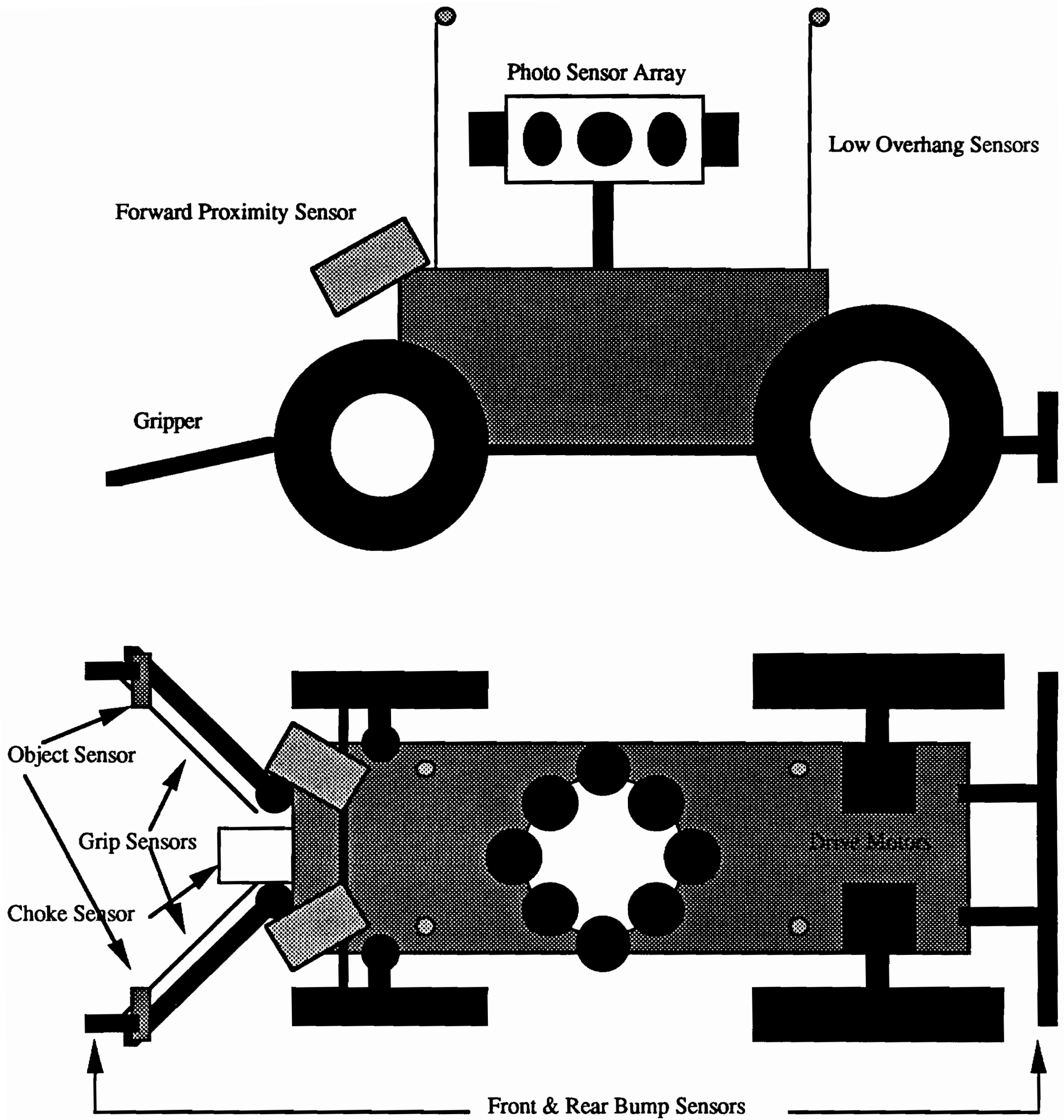


Figure 5.1-1: The "Tooth" Robot, side and top views.

5.1.2 Experimental Setup

The robot's objective in this experiment was to explore the perimeter of a test course, pick up the small cylindrical objects located there, and deposit them by a beacon. Figure 5.1-2 shows the layout of the test course. The shaded areas are obstacles and/or walled off areas (e.g., a desk, a trash can, etc). The white circle is the beacon, a 150-watt light bulb. The small black circles are the objects to be retrieved (known as "Tooth toys"). The tooth toys are small cylinders made of plastic and foam. They are about two inches in diameter and weigh approximately an ounce.

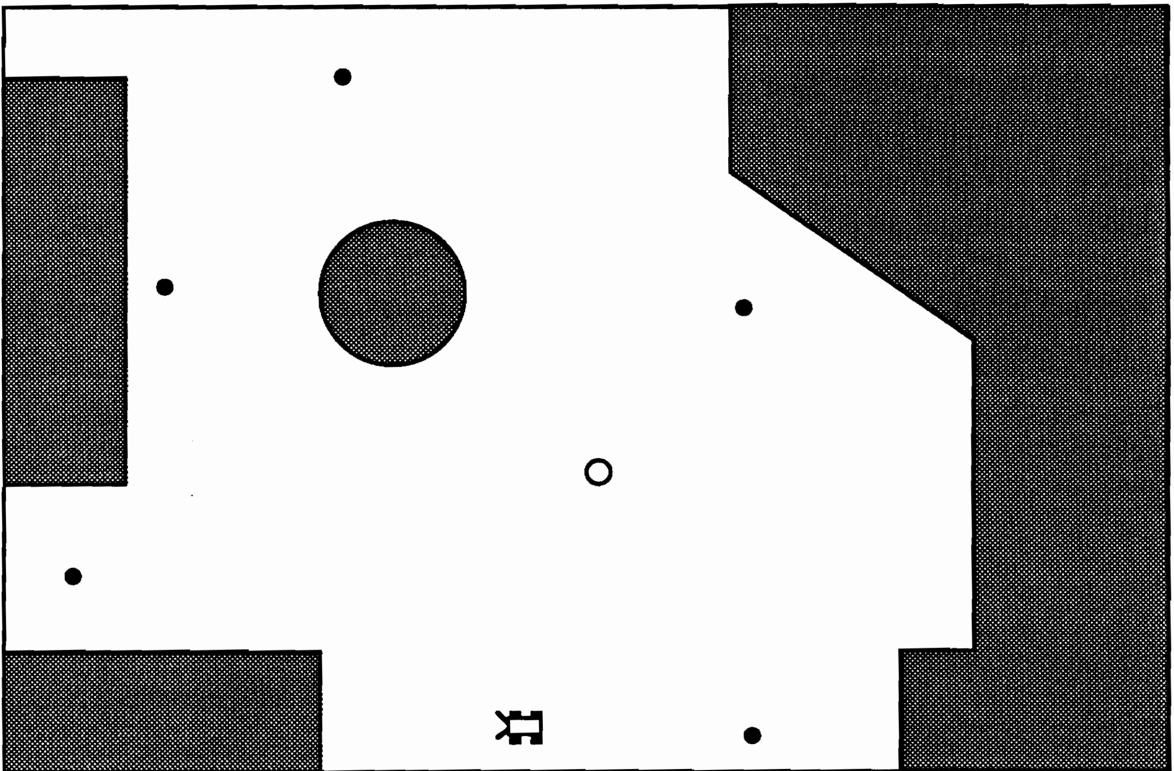


Figure 5.1-2: The layout of the test course at the start of one experiment.

5.1.3 The Program

The program which controlled the robot was distributed over its two processors. (See figure 5.1-3.) The first processor was connected to the CdS photocells and the controls for the gripper, and the second was connected to the drive and steer motors, as well as the bump sensors and infrared proximity sensors. (These processors are therefore referred to as the grasp processor and the drive processor respectively.) Communications between the two processors take place over a serial link. This fact is completely transparent to the program. The serial link simply transmits data from a channel on one processor to a corresponding channel on the other.

The drive processor contains modules for avoiding obstacles and recovering from collisions. It also contains a module which allows the robot to escape dead ends. This is accomplished as follows: when Tooth's path is blocked (detected by both proximity sensors firing, or the front bump/overhang sensors firing) then the robot backs up for a short amount of time. The amount of time the robot backs up is recorded in a state variable. If, when the robot starts to move forward again, it is forced to back up a second time before it has moved forward longer than it has moved backwards then the amount of time it moves backwards is increased. This eventually causes the robot to back up enough to escape the dead end. This together with random perturbations (e.g. wheel slippage) eventually cause Tooth to find its way out of most dead ends.

The grasp processor program has modules for picking up a toy, dropping the toy when near the beacon, and directing the vehicle to or away from the beacon. This last module uses the photocells to determine the direction to the beacon, and also calculates the turning direction and the drive speed and direction. These values are then passed via the interprocessor communications channel to the drive processor which uses them as input to its modules.

The grasp processor also implements a rudimentary form of failure recovery. The vehicle has contact sensors on the inside of its gripper that can tell it if it has successfully grasped an object. If the robot fails to pick up an object successfully after several attempts the robot is made to back away from the object. This prevents the robot from endlessly attempting to pick up objects that it cannot grasp.

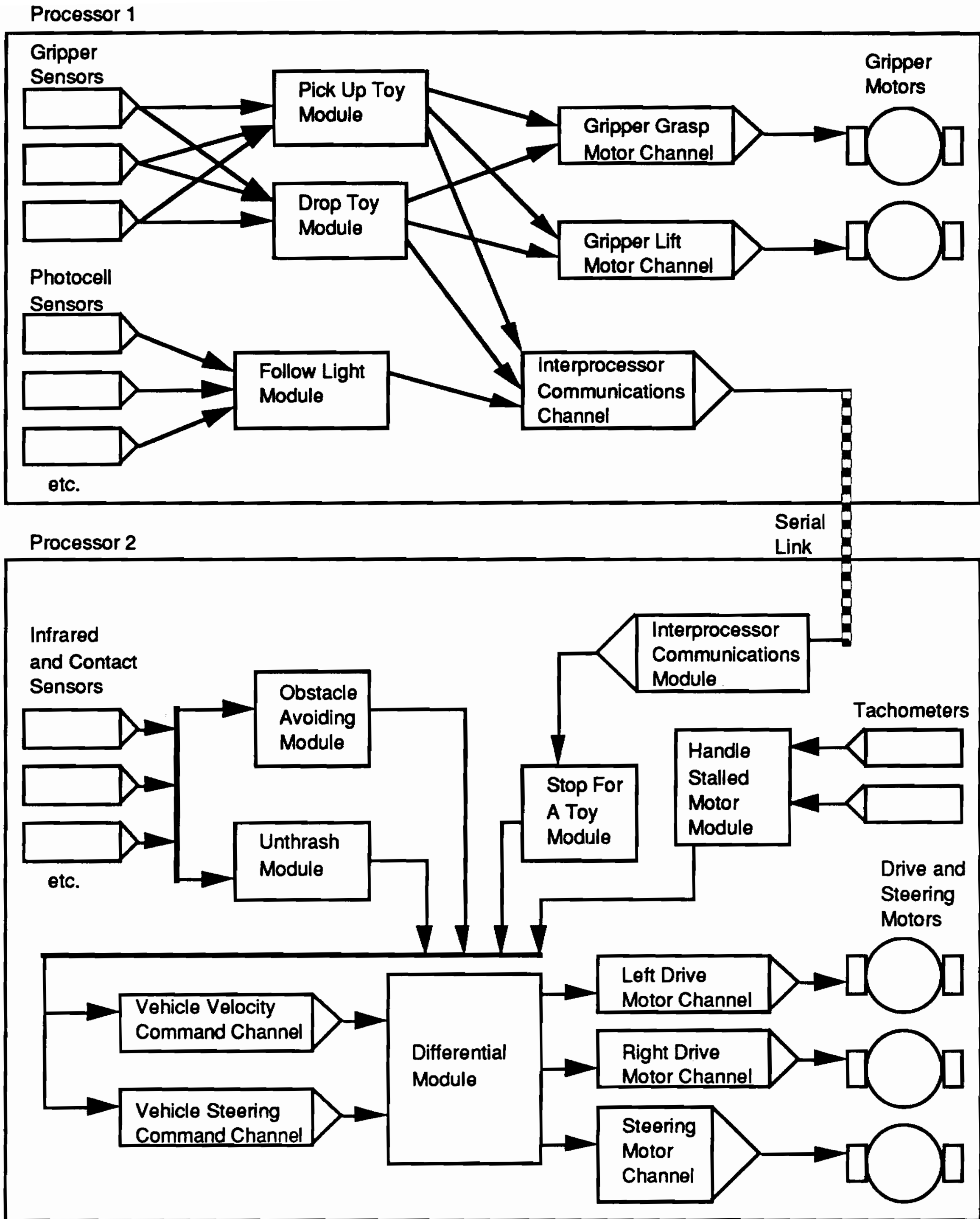


Figure 5.1-3: A diagram of the object-collecting program.

Even this simple example demonstrates some of the design principles of the ATLANTIS methodology. The control system is built up in layers which operate at successively higher levels of abstraction. At the bottom is a layer which connects directly to the vehicle's actuators. The next layer implements a collision-free motion activity. This layer is guided at a higher level of abstraction by the third layer which tells the second layer which way to try and go, and when to stop so that the robot can try to pick up an object. Note that these layers do not follow the subsumption methodology, despite the superficial resemblance. Each layer operates at a higher level of abstraction than the one below, and only the bottom layer interfaces directly to the robot's actuators. In subsumption, all layers operate at the same level of abstraction, and all interface directly to the robot's actuators. Tooth's control layers are described in detail in the following sections.

Layer 1

This layer consists of a single module, the differential module, and six channels. Four of these channels are actuator channels which interface to the robot's two drive motors and its steering motor. (One of these channels is omitted in the figure.) Controlling these motors is quite complex. The left and right drive motor velocity must be controlled according to the robot's steering to provide the correct differential velocities. Furthermore, because of the way the hardware is set up, the direction of the drive motors is controlled separately from the magnitude of the velocity. All of these complexities are completely hidden from the rest of the system by the bottom layer. All subsequent control layers can control the speed and direction of the vehicle directly by controlling the two interface channels in the bottom layer. These channels form virtual actuators, and hide many of the details of the system behind a layer of abstraction.

Layer 2

This layer implements collision-free navigation. It consists of four modules. The first one causes the vehicle to veer away from obstacles detected by the robot's infrared proximity detectors and contact sensors. If one infrared sensor is activated, the robot turns in the other direction. If both infrared sensors or a forward contact sensor is activated, the robot backs up. If a rear contact sensor is activated while backing up, the robot stops. All

of these sensors are connected to timers which cause the effects of an obstacle to persist for a while after the sensory stimulus is removed.

The second module handles motor stalls. If a motor stalls while the robot is moving forward, it backs up. If a motor stalls while backing up, the robot stops. Again, these effects are connected to timers. If the robot's motors stall while backing up, it will stop only for a few seconds, and then try again in the hopes that whatever was blocking it might have gone away.

The third module gets the robot out of futile loops. This module counts the number of times the robot changes direction in a given period of time. If a threshold is exceeded, this module forces the robot's steering motor into a random configuration. The idea is that if the robot is moving back and forth, a random steering command might get it out of the loop. In practice this has proven quite effective. In many hours of use, Tooth has never gotten stuck in a futile loop for more than a minute or so (except in the presence of obstacles that the robot's sensors could not detect).

The fourth module takes commands from the next control layer and attempts to move the robot in response to these commands. The commands consist of a nominal direction and speed, and are produced by the third layer in response to the robot's task requirements.

These modules interface to the first layer by means of two prioritized channels which set the robot's speed and direction. The futile-loop module interfaces only to the direction channel, and has the highest priority. The external-command module has the lowest priority. The obstacle avoidance and stall behaviors have the next highest priorities.

Layer 3

The third layer controls the vehicle at a very high level of abstraction. This layer monitors the beacon and gripper sensors, and tells the second layer which direction to try to go. It also tells the second layer when to stop so that the vehicle can pick up a toy. The gripper itself is also controlled by this layer. The module which picks up a toy is interesting because it implements a simple example of failure recovery. If the vehicle tries several times without success to pick up the same object, this module will give up and make the vehicle back away from the object so that it can try again somewhere else.

This layer controls the vehicle's speed and direction at a very high level of abstraction, providing only a nominal velocity and speed as guidance to the second layer. All of the details of obstacle avoidance are hidden beneath a layer of abstraction.

5.1.4 Results

There are no formal quantitative results from these experiments. However, Tooth has been run many dozens of times in a variety of environments, and consistently and reliably performs the task for which it was designed. The are only two serious problems with Tooth's behavior, both of which are the results of sensor deficiencies. First, it has no way of sensing toys remotely, and therefore must sometimes wander blindly for a long time before coming across one. Second, it has no way to sense non-somatic obstacles such as wires or holes, and therefore cannot deal with them.

Figure 5.1-4 shows the start of a typical run. The figure was generated by hand from a video tape of the run. The robot followed the perimeter of the test course until it encountered a toy near the lower left hand corner. It picked up the toy, thrashed in the corner for a while, and then dropped the toy off near the light. The events shown in figure 4 required less than one minute. Tooth was able to acquire all the toys and deposit them by the light within eleven minutes with only minor human intervention. The human intervention was required in cases where Tooth would nudge a toy out of its original position, either because its gripper was already holding a toy, or because the object was contacted by the tip of the gripper or by one of Tooth's wheels. The human intervention consisted only of moving the toys back into their original location.

Finally, it was found after the experiment was complete that the left-facing CdS cell was inoperable. This explained why Tooth sometimes took slightly longer than expected to turn towards the beacon. It also lends some weight to the claim that reactive control is robust in the face of hardware failure.

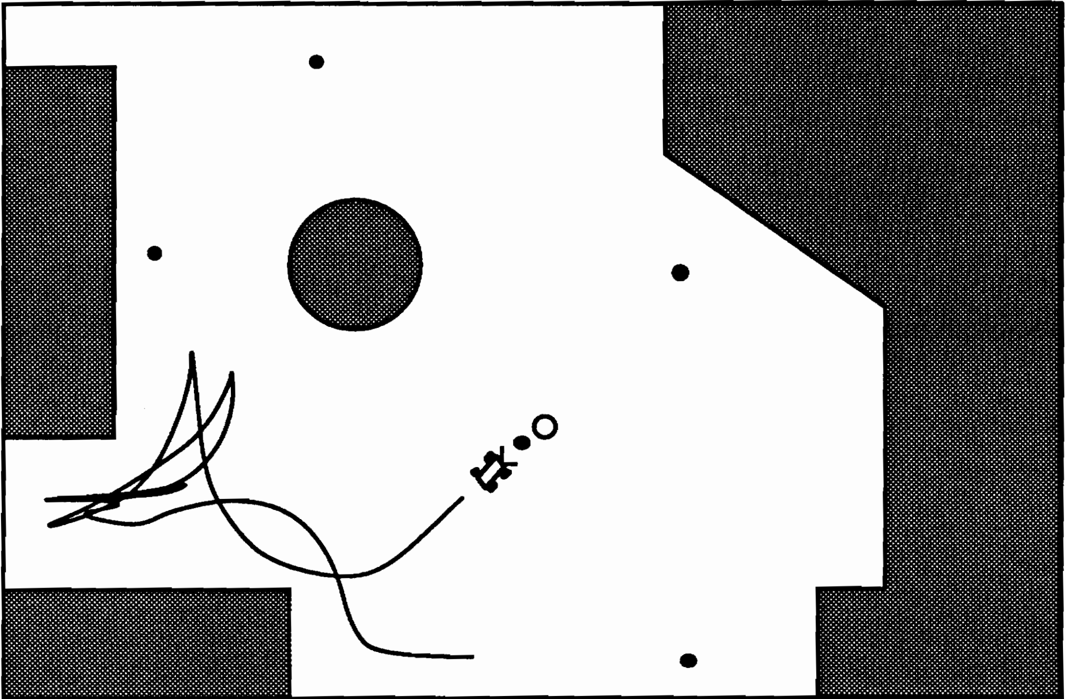


Figure 5.1-4: The test course, and the path taken to retrieve the first object.

5.2 Toto

Toto [Mataric90] is an indoor robot which was used to demonstrate that ATLANTIS could perform a complex navigation task by generating a sequence of primitive activities such as wall-following, dead-reckoning around obstacles, and going through doors. The control layer was programmed in ALFA and ran on a 68000-based computer on board the robot. The sequencing layer was written in LISP and run on a Macintosh II. There were several advantages in putting the sequencing layer off-board. First, it allowed the interface between the sequencing and control layer to be rigidly enforced. Second, it allowed the two layers to run truly asynchronously. In fact, on a number of occasions the serial cable became disconnected inadvertently, and the robot continued on its way for some time before this was discovered.

No deliberative layer was implemented on this robot; the planning which this layer would have performed was done by hand and given to the system *a priori*. The plan that was given to the robot is one which can be easily generated by straightforward and well

understood techniques (e.g. [Erdmann90], [Malkin90], [Hendler90]), and so this did not sidestep any relevant issues.

5.2.1 The Robot

Toto [Mataric90] is a small cylindrical indoor robot. It consists of a custom enclosure mounted on a Real World Interfaces twelve-inch synchrodrive base. Toto's only sensor modality is a ring of twelve sonars [Biber80] arranged at equiangular positions around the robot and located about forty cm. off the ground. (See figure 5.2-1.) Toto has a synchrodrive mobility system which allows it to turn in place. Computation is provided by an on-board 68000 processor with about 500K of memory, though only a small fraction of this was actually used. The computer had a minimal operating system in read-only memory which provided device drivers for a number of serial communications ports which were used to communicate to peripheral processors which controlled the sonars and the motors.

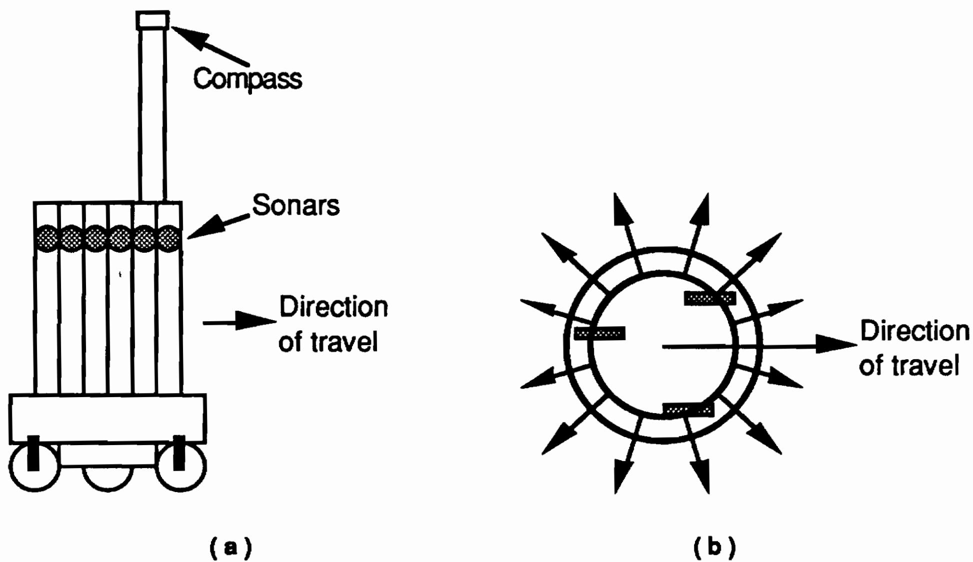


Figure 5.2-1: Toto in profile (a) and from above (b).

The sonars have a beam width of about 30 degrees. Coverage was therefore complete around the robot's perimeter. However, because the sonars were 40cm off the ground,

low-lying obstacles could not be detected. The sonars were controlled by a peripheral processor which polled the sonars in pairs. Each pair was polled in turn. Polling all twelve sonars took about two seconds. The shortest resolvable distance was about a foot, that is, an object a foot away would return the same reading as an object butted against the vehicle. The resolution beyond this distance was about 15cm, and accuracy decreased dramatically beyond about 4 meters. The sonars also suffered from a characteristic failure mode known as specular reflection. If the incident angle of the sonar beam on a smooth surface was greater than approximately half the beamwidth of the sonar the beam would be reflected as if the surface were a mirror, rendering it essentially invisible.

5.2.2 Safety Modules

The control layer was developed first, beginning with a safety module which would stop the robot if it was in danger of colliding with an obstacle. This turned out to be surprisingly non-intuitive. At first glance one might suspect that it would be sufficient to monitor the four front sonars (or perhaps even the front two) and stop the robot if any of them registered a minimal-distance reading. This turned out to be insufficient. There are two major situations in which this scheme fails. The first is when the robot approaches a convex corner, and the second is when it approaches a smooth surface such as a wall at an oblique angle. (See figure 5.2-2.) Specular reflections render such obstacles invisible to all four forward-looking sonars.

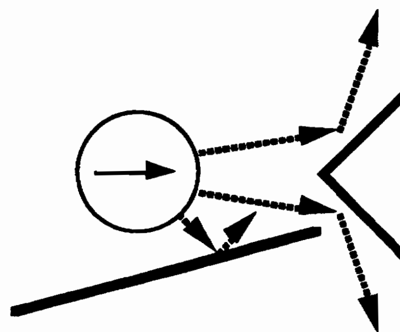


Figure 5.2-2: Two failure modes for obstacle avoidance using forward sonars only.

In order to deal with convex corners additional sensors are needed. The information to distinguish corners from open space is simply not available. Fortunately, corners are not a serious problem in practice. For the corner to be truly invisible to the sonar it must be exactly in between the two forward-looking sonar beams. An extra sonar pointing straight ahead or a set of infrared proximity sensors would probably solve this problem.

Approaching a flat surface at an oblique angle, on the other hand, tends to happen quite often. To prevent collisions therefore all six forward-looking sonars had to be monitored. If any of these six sonars showed a minimal-range reading, the robot was made to stop. This insures that the robot will not collide with obstacles provided the sensors can detect them. (In many weeks of use, the only situations in which the robot got into trouble involved low-lying obstacles below the field of view of the sonars. These were uncommon enough that the robot could safely roam the hallways unattended for extended periods of time.)

Because the sonars were only polled once every two seconds the robot had to limit its speed in order to move safely. The maximum safe speed is the speed which moves it no further than the distance to the nearest known obstacle before a new sonar reading is taken. Thus, if the distance to the nearest obstacle is d, the maximum safe speed is d/(2.0 seconds). At this speed, the robot will just reach the obstacle when the next sensor reading is taken. If everything works perfectly, this reading will be 0 and the robot will stop instantly gently abutting the obstacle. To accommodate reality the robot was not allowed to move at more than d/(4.0 seconds). This permits the robot to move only half the distance to the obstacle between sensor cycles, resulting in the robot smoothly slowing down as it approaches an obstacle. In practice the robot's velocity does not change smoothly because the two-second sampled signal is unfiltered. However, it does in fact cause actual collisions to be exceedingly rare despite noisy sensor data. The robot would, on rare occasions, squeeze itself into very tight spots, but collisions with walls or furniture did not occur.

To detect collisions with low-lying unseen obstacles the robot's drive current was monitored. The robot's motor controller was velocity-controlled, so the drive current would increase if it collided with an obstacle which it could not push out of the way. If the drive current exceeded a threshold, the robot was made to stop. This functionality was rarely needed during the course of the experiments.

5.2.3 Following Walls

Toto's dead reckoning is quite good. On tile floors the error can be less than 1% of distance travelled over distances of less than 100 meters. Dead reckoning error is not linear. Because the robot experiences errors in orientation as well as position, the dead reckoning errors tend to accumulate as the square of the distance travelled. Thus, relying on dead reckoning, even very good dead reckoning, can get the robot hopelessly lost.

In order to obtain results which did not depend on dead reckoning, an artificial dead reckoning error was introduced by misaligning one of the robot's three wheels. This produced an unpredictable angular error of up to 10 degrees per meter travelled. The actual error varied greatly with the robot's actual orientation and the direction of the carpet nap. The carpet nap independently introduced a linear displacement error biased in the direction of the nap of about 5% of distance travelled. However, this was insignificant compared to the angular error caused by the misaligned wheel. Effectively, dead reckoning was completely unreliable beyond about three meters. The robot could dead reckon across the hallway or past a door, but going across a large room blindly was out of the question.

Under these circumstance Toto could only get from place to place by following the walls like a blind person [Haber90]. Wall-following was implemented as a primitive activity. The basis of the wall-following activity was the specular reflection property mentioned earlier as a shortcoming of the sonar sensor. Because there are two sonars on either side of the robot aligned at 15 degrees from the perpendicular they will both return true range readings only when the robot is very nearly aligned with the wall. (See figure 5.2-3.) Thus, we can align the robot to the wall by checking which side-looking sonar is giving a specular reading and slowly rotating the robot in the appropriate direction as it moves.

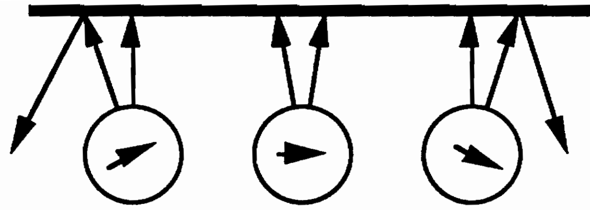


Figure 5.2-3: Aligning to the wall by looking for specular reflections.

The resulting system tends to keep the robot aligned to the wall, but does not correct for accumulated lateral drift. This is a serious problem because the specular-reflection trick only works if the robot is close enough to the wall (within about 1.5 meters), but moving too close to the wall will force the robot to stop or risk a collision. Thus, the robot must check to see if it is moving too close to or too far from the wall and make corrections accordingly.

In the following sections we will examine in detail the process of designing the module to control wall following. The process involves two steps. First, the control law itself is derived. Next, it is augmented with mechanisms for detecting failures.

5.2.3.1 Deriving the Control Law

Because the sonars are noisy and inaccurate it is not possible to use their data to perform feedback control with any appreciable precision. Because of this, there is no point in trying to construct a precise control law; computations should never be performed to greater precision than the input data. Therefore, let us approach the problem qualitatively and consider each sonar as returning only one of three possible readings: short, nominal, and long. A short reading means that the robot is too close the wall. A medium reading means the robot is about where it should be. A long reading means that the robot is either too far away or is in specular reflection. Thus, we can divide the phase space of the two side-looking sonars into nine regions as shown in figure 5.2-4.

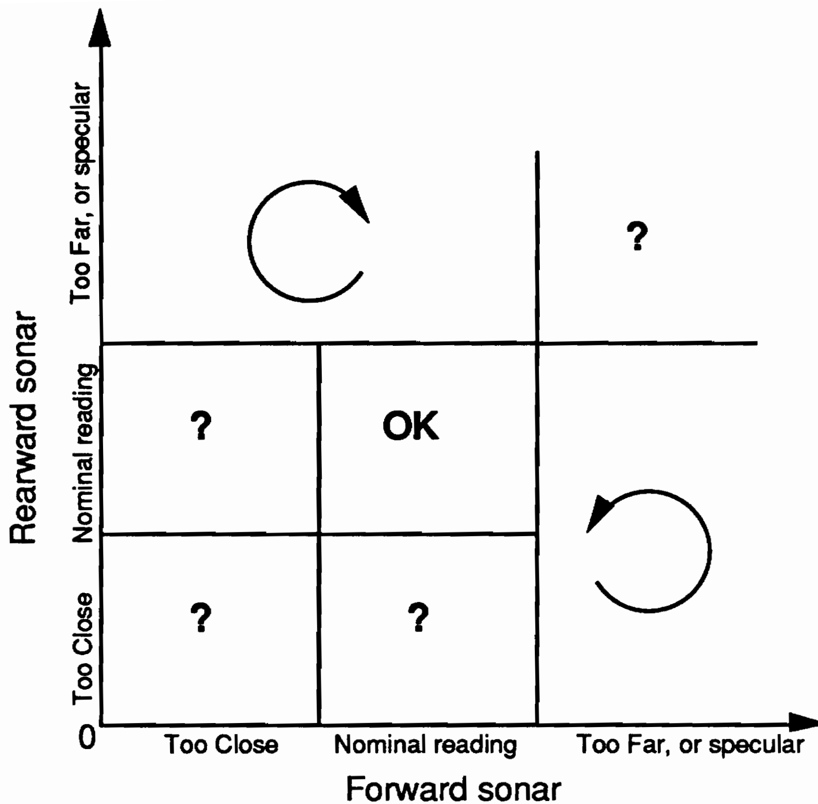


Figure 5.2-4: The phase space of two sonars and a partial description of the alignment module's transfer function.

Let us assume for the moment that a long reading indicates a specular reflection. When the rear sonar shows a long range reading, we need to rotate clockwise to align with the wall (assuming we are following a wall on the left - for following on the right everything is reversed). When the front sonar shows a long reading we need to rotate counterclockwise. If both sonars show nominal readings then the robot is exactly where it should be - aligned to the wall and at the proper distance. It is not yet clear at this point what to do in the other regions.

In particular, it is not clear what should be done when both sonars show a long reading. This could indicate a number of things. It could mean that the robot has drifted too far from the wall. It could mean that the robot has encountered a door. Or it could mean that both sonars are returning specular reflections. The latter situation could be caused by three things. It is possible for both sonars to reflect simultaneously in opposite directions (i.e. one to the front of the robot and one to the rear) if the wall is exceptionally

smooth and the robot is closely aligned to it. On the other hand, two long readings could mean that the robot is badly misaligned with the wall, and both sonars are reflecting off the wall in the same direction. Finally, it could mean that the robot is by a door. Because the corrective action required in each of these cases is different, we must either be able to distinguish between them, or insure that the ones that we cannot distinguish never arise.

The latter approach turns out to be easier and more reliable. It requires quite a severe misalignment - 30 degrees or more - for both side sonars to reflect specularly off the wall in the same direction. Thus, if we can insure that our alignment error never exceeds 30 degrees we can safely assume that two long range readings indicates either that the robot is too far away from the wall, or that the robot is at a door, or that the robot is aligned to a smooth wall and both sonars are reflecting, but in opposite directions.

Let us ignore doors for the moment; we shall return to them shortly. That leaves us with only two possibilities: either the robot is too far from the wall, in which case it should turn counterclockwise (assuming it is following the left wall), or the robot is aligned to the wall, in which case it should not turn at all. Unless we can somehow distinguish between these two situations, it would seem that we are at an impasse. Note, however, that if both sonars are reflecting, then turning to the left will almost immediately bring the front sonar into a position where it will once again return a true range reading. This will bring the robot back into a region of phase space where it turns clockwise, immediately correcting the erroneous move. Thus, we can safely turn counterclockwise when both sonars return long range readings (if we ignore doors).

We can also fill in the region where both sonars return short range readings. In this case the robot is too close to the wall and should turn clockwise. The robot will not overcompensate because the forward sonar will go into specular reflection before the robot turns too far, causing the alignment module to turn it back towards the wall. This leaves us with the situation in figure 5.2-5.

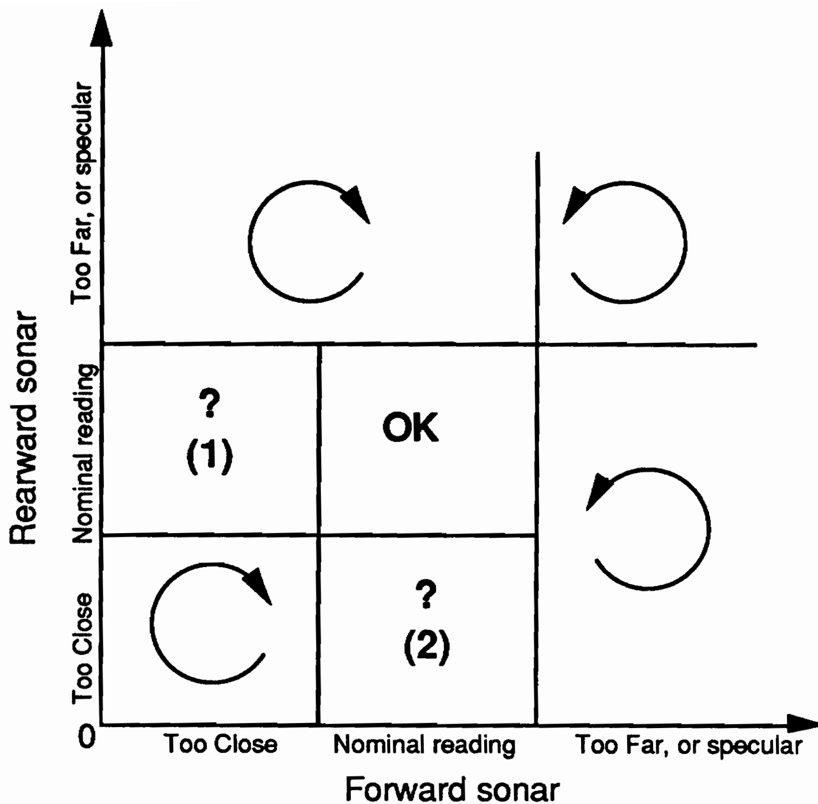


Figure 5.2-5: The second step in the construction of the alignment module transfer function.

We now have two regions left to fill in. These regions are fairly peculiar in that both sonars seem to be returning true range readings, but they disagree significantly on the distance to the wall. We might be tempted to conclude that this situation will never arise in practice and not worry about it, but in fact it tends to come up quite often because of sensor errors. Fortunately, there is a certain amount of regularity to the errors on this type of sonar: erroneous readings are almost always too long rather than too short. Thus, we can assume that the error is with the longer of the two readings and treat these two regions the same as the lower-left region, i.e. turn clockwise. The final transfer function is shown in figure 5.2-6.

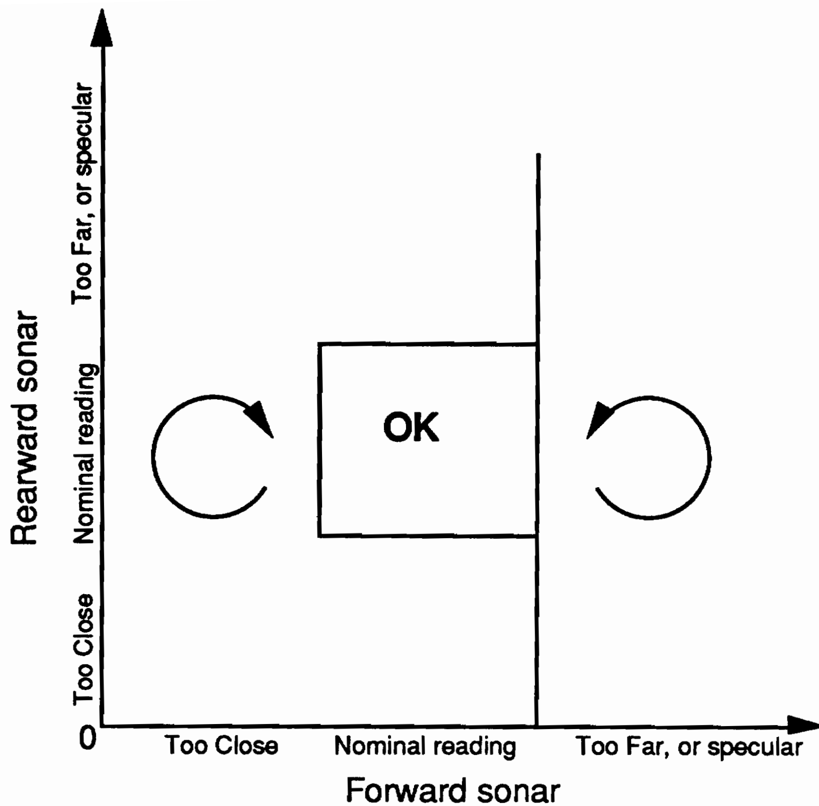


Figure 5.2-6: The final transfer function for the alignment module.

This transfer function can be implemented with the following simple piece of ALFA code:

```
(cond ( (> forward-sonar-channel nominal-reading)
        (drive turn-motor-channel counter-clockwise) )
      ( (< forward-sonar-channel nominal-reading)
        (drive turn-motor-channel clockwise) )
      ( (or (> rear-sonar-channel nominal-reading)
            (< rear-sonar-channel nominal-reading) )
        (drive turn-motor-channel clockwise) ))
```

This code assumes that `clockwise`, `counter-clockwise` and `nominal-reading` are constants which have been defined elsewhere in the program. This code is specific to following walls on the left. It is a trivial matter to generalize it to follow walls on the right.

5.2.3.2 Discussion and Analysis

This transfer function has a number of remarkable characteristics. First, it turns out to be astoundingly simply, almost trivial. Second, it seems to depend almost exclusively on the value of the forward sonar. Let us see if these results are reasonable.

First, let us examine the behavior of the robot informally. Figure 5.2-7 shows the phase space diagram with a schematic representation of the actual state of the robot in the various regions. A brief comparison of this figure with figure 5-6 will show that the behavior we have prescribed is not unreasonable. Turning clockwise in the lower middle region is, perhaps, a bit questionable since that tends to turn the robot away from being aligned with the wall. However, the robot does need to get further away from the wall in this case, and so turning clockwise is not completely outrageous. Besides, the robot will stop turning clockwise when it reaches the point where the front sonar starts to return specular reflections, and so the robot will not overcompensate.

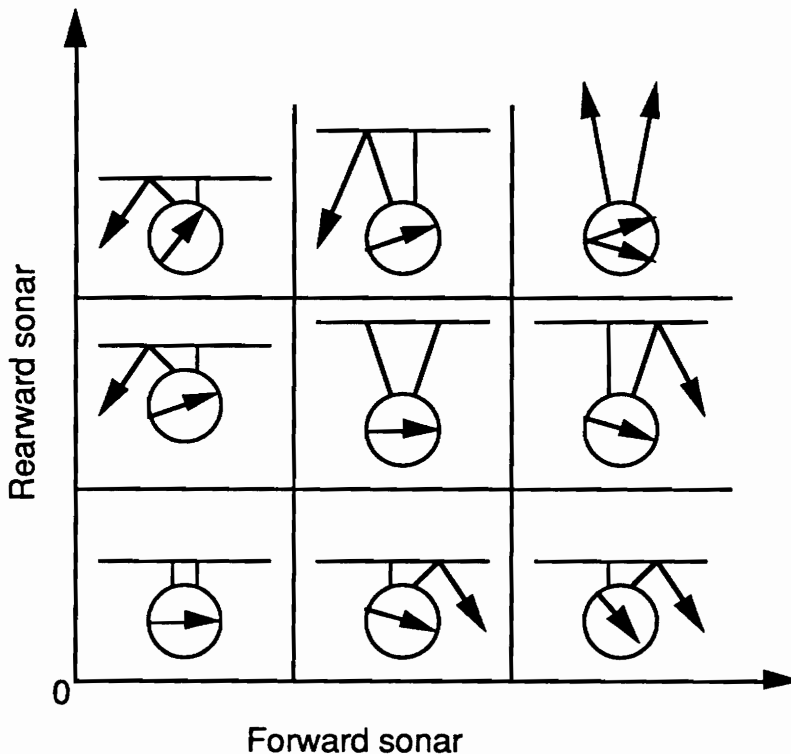


Figure 5.2-7: The state of the robot in each region of sensor space.

There is a more formal argument that this transfer function is correct, which also sheds some light on why the forward-facing sonar is more important in determining what to do than the rear-facing sonar. Let us simplify the transfer function so that the rate of rotation depends on the front sonar reading only. Let r designate the sonar reading and let the rate of rotation be governed by the following control law:

$$d\theta = \sigma(r - r_0)\, dt \qquad (5\text{-}1)$$

where σ is some monotonic control function which crosses the origin. Now, let y be the distance from the robot to the wall, and let the angle between the sonar and the direction of travel be ϕ. (See figure 5.2-8.) Then the equations of motion for the robot are:

$$dx = v \cos(\theta)\, dt \qquad (5\text{-}2)$$

$$dy = -v \sin(\theta)\, dt \qquad (5\text{-}3)$$

where v is the robot's velocity, and r, the sonar reading, is given by:

$$r = y / \sin(\theta + \phi) \qquad (5\text{-}4)$$

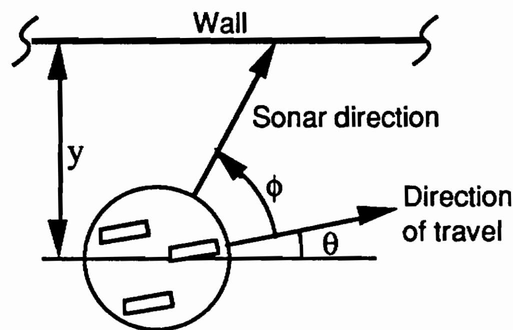


Figure 5.2-8: Proving the stability of the alignment module transfer function.

The motion of the robot, then, is governed by equations (5-1) and (5-3), a system of two non-linear differential equations in two unknowns, y and θ. Solving these equations

can only be done numerically. However, we can show that the system is unstable when v is small and $(\theta + \phi)$ is greater than $\pi/2$. Differentiating 5-4 gives:

$$dr = [\sin(\theta + \phi)\, dy - y \cos(\theta + \phi)\, d\theta] / \sin^2(\theta + \phi) \qquad (5\text{-}5)$$

Substituting (5-1), (5-3) and (5-4) in (5-5) and rearranging terms yields the differential equation:

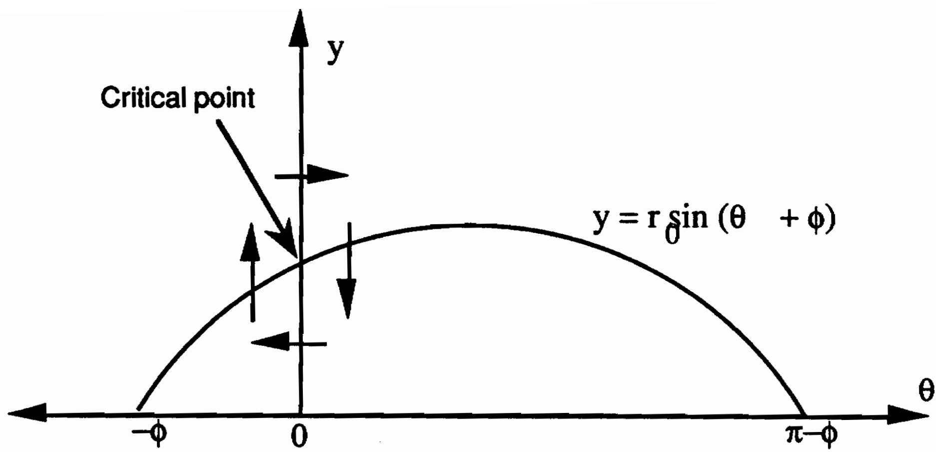
$$dr/dt = [\, -v \sin(\theta) - r\, \sigma(r - r_0) \cos(\theta + \phi)\,]\ / \sin(\theta + \phi) \qquad (5\text{-}7)$$

Now we assume that v is small, allowing us to ignore the first term on the right hand side of the equation. This gives us the following homogeneous non-linear first-order equation in r:

$$dr/dt + [\cot(\theta + \phi)\, \sigma(r - r_0)]\, r = 0 \qquad (5\text{-}8)$$

Because σ is monotonic and sign-preserving, this equation is unstable when $\cot(\theta + \phi)$ is negative, i.e. when $(\theta + \phi)$ is less than 0 or greater than $\pi/2$. Thus, for sonar-based wall following, using a forward-looking sonar for feedback is a necessary (but not a sufficient) condition for stability when the velocity is small compared to the rate of rotation. (This was first observed informally in [Miller85].)

We can also show that the system is stable when $\phi<\pi/2$ by drawing a trajectory diagram. (See figure 5.2-9a). The trajectory diagram is divided into four regions by the line $\theta=0$ and by the curve $y=r_0 \sin(\theta + \phi)$. These intersect at a critical point corresponding to where the robot is exactly aligned to the wall at the correct distance. To the left of the line, dy/dt is positive; to the right dy/dt is negative. Above the curve, $d\theta/dt$ is positive; below it is negative. These directions are indicated with arrows in the diagram. Intuitively, then, it appears that the solution of the equations should circulate in a stable trajectory around the critical point.



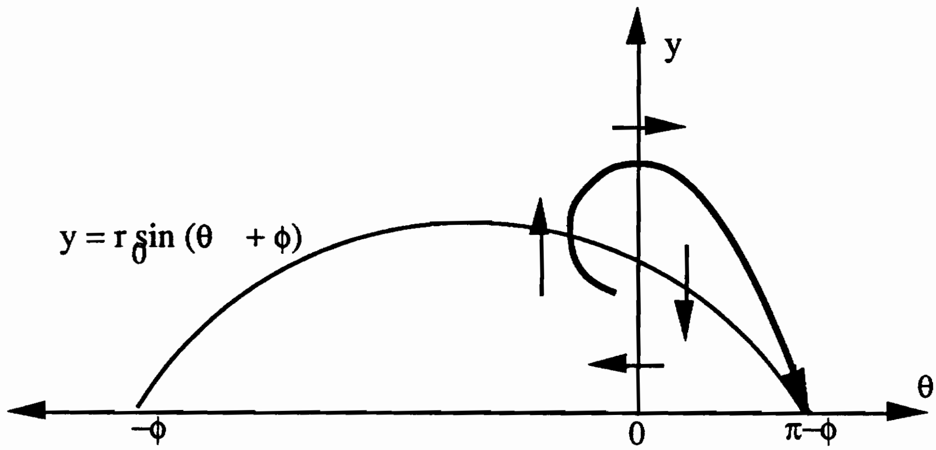


Figure 5.2-9: The phase diagram for equations (1) and (3) (a) when $\phi<\pi/2$ and (b) when $\phi>\pi/2$.

We can show this analytically by letting σ be a function which linearizes $y/\sin(\theta + \phi)-r_0$. In this case the equations of motion become:

$$dy/dt = -v \sin(\theta) \tag{5-9}$$

$$d\theta/dt = c(y-y0) \tag{5-10}$$

where c and y_0 are constants. This system is identical to one describing the motion of an undamped pendulum and is known to be stable [Boyce77].

The phase diagram also illustrates why the system is unstable when $\phi > \pi/2$ and v is small. (See figure 5.2-9b.) In that case the curve is shifted to the left, and the resulting system admits unstable trajectories such as the one shown by the heavy line. However, if v is large, the trajectory tends to move up and down more than it moves side to side, keeping the system stable. This makes intuitive sense if one considers that a rear-looking sonar provides negative feedback for y but positive feedback for θ. The negative feedback is proportional to the velocity. Thus, if the velocity is large, the negative feedback overrides the positive feedback and the system is stable.

Deriving necessary conditions on σ for stability is quite complex -- and quite beside the point. Even if we could derive such conditions, meeting them would still not guarantee that our robot would work. This is because there are a great many unwarranted assumptions that went into writing the equations of motion in the first place. Among these are: linear motor control, perfect sensor data available at infinite bandwidths, a control mechanism operating at infinite bandwidths, and others. These are precisely the assumptions that this thesis wishes to discharge!

This underscores the point made earlier about the difficulty of formally proving anything about ATLANTIS. A formal analysis of even this trivial case - using only a single sensor and making numerous simplifying assumptions - yields a system of complex non-linear differential equations. Discharging the simplifying assumptions may begin to tax our ability to even write down the equations, let alone solve them.

Even in cases where we can do the math, we must be careful how much we trust the results. This point was dramatically illustrated while testing the wall-following algorithm. An early version of the algorithm was found to fail consistently along a particular stretch of wall, even though it worked reliably along other walls, and even other parts of the same wall. It was eventually discovered that this was due to the fact that the tiny grooves between the drywall panels had been filled in with caulk along the troublesome section of wall, resulting in different specular reflection properties. Any predictive theory of the robot's behavior would have to take into account the effects of these millimeter-sized grooves in order to be accurate.

Let us return now to the practical considerations of stabilizing our wall-following module. Because the bandwidth of the sensor data is quite small (about 1/2 Hz), both the rate at which the robot moves (v) and the rate at which it rotates to correct its heading (a parameter of σ) must be carefully controlled. If the robot moves too fast it could move too near or too far from the wall before the distance-maintaining module had a chance to correct it. If the robot changes its heading too fast it could overcompensate and end up badly misaligned in the other direction. On the other hand, if the robot corrects its heading too slowly then it might not be able to keep up with the rotational error introduced by the misaligned wheel on the carpet.

There are two independent factors we must consider. First, the rates must be slow enough that the discrete updates of the robot's trajectory approximate the behavior of the continuous system enough that the robot will be stable. Second, the ratio between the two rates must be such that the trajectory stays within the stable region of phase space.

The primary restriction on the stable region is that θ must lie between $-\phi$ and $\pi/2-\phi$. On Toto, ϕ is 75 degrees, so θ must never be allowed to be larger than 15 degrees. To be safe, we should choose a rate which allows at least two sensor readings to be taken before the robot can rotate through this angle, allowing it a chance to recover should an erroneous reading occur at an inopportune moment. Thus, the robot should rotate no faster than 7.5 degrees every four seconds, or about two degrees per second.

The maximum allowable velocity is that which causes the maximum allowable lateral error during the maximum possible heading error. The maximum heading error occurs when the robot turns constantly from q=0 until $\theta=\theta_{max}=15$ degrees. At this point the robot must begin turning the other way (if all our assumptions are satisfied). If we assume that the absolute value of the angular velocity is constant, then the heading error induced in this case is $2\int v \sin(\theta)d\theta$ evaluated from 0 to 15. Solving for v yields a number which is much larger than the maximum safe speed for avoiding obstacles. Thus, we can ignore this limit in favor of the more conservative one.

5.2.3.3 Providing Cognizant Failure

There is one last issue to be resolved. We have assumed throughout this discussion that the robot will not encounter doors. In fact, the algorithm will fail if it encounters a door because the long readings will cause the robot to rotate towards the door. By the time

the robot has passed the door it will be dangerously close to the wall, perhaps even in danger of colliding with the door jamb.

One thing we can do at this point is to simply fail and let the sequencing layer sort things out. However, we can do better. A very simple trick can let the robot distinguish doors from double specular reflections. The key is to note that double specular reflections can occur only if the robot is very nearly aligned with the wall. Thus, only a small perturbation in the robot's alignment will cause one or the other of the two sonars to return a true range reading. Thus, if the robot turns a little bit and checks again before failing, it can distinguish between doors and double reflections.

Implementing this is trivial because we have already arranged for the robot to turn to the left when both sonars return long readings[3]. Thus, all we need to do is to install a small delay when the robot first detects two long readings. This gives the robot enough time to turn and take another set of sonar readings. If one of these readings is not long, then the robot simply proceeds. Otherwise, the robot stops and reports a wall-following failure due to encountering a door.

On Toto it was necessary to make one small modification to this procedure. Because the sonars were fired at such a low rate it was possible for the robot to traverse most of the width of a door in the time required to complete the second sonar scan. Thus, the robot was made to stop while waiting for the second set of sonar readings to confirm the existence of a door. This procedure turned out to be extremely reliable.

The final problem is how to tell when something completely unexpected goes wrong, e.g. the robot is not next to a wall. This was done by measuring the distance from the last place the robot was able to successfully align itself to the wall. If the robot travelled more than a certain distance (2 meters) without an alignment, a failure was signalled to the sequencing layer. This distance was chosen because the accumulated dead reckoning errors at this distance were still small enough for the robot to potentially recover from the failure by dead-reckoning to some other nearby wall.

---

[3]Assuming, as usual, that we are following left walls. Obviously all this generalizes to following right walls also.

5.2.4 Going Through Doors

To get around in an indoor environment a robot must be able to go through doors. This turns out to be quite straightforward. The wall-following activity tends to leave the robot fairly close to the center of the door, but not close enough for the robot to reliably dead-reckon its way through the door. Furthermore, to insure safety the robot may not move when any of its forward sonars are showing minimal range readings. In other words, the robot cannot move if there is anything within about 35cm in front if it or to the sides. Since the robot is about 35cm wide and doors tend to be just under a meter wide this results in a tight squeeze.

Fortunately, this turns out not to be a problem in practice. To get through doors the robot starts by heading in the general direction of the door until it is blocked. It then begins to scan back and forth within a 180 degree angle of (its current best guess at) a line perpendicular to the door. Whenever it is not blocked it moves forward. This procedure will tend to "wiggle" the robot through the door. (See figure 5.2-10.) It is quite reliable in practice, and it has the fortuitous side effect of leaving the robot very nearly aligned with the center of the door. (cf. [Connell89])

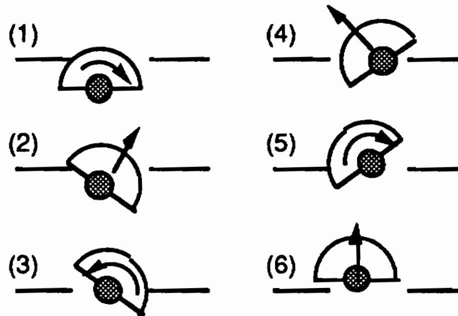


Figure 5.2-10: Squeezing through a narrow door. The shaded circle is the robot, and the semicircle is the collision danger zone.

5.2.5 The Sequencing Layer

The sequencing layer contained seven task schemas. Three of these were simple interfaces to the primitive activities for dead reckoning, wall-following and going through doors. The other four were more complex task schemas with multiple methods for dead reckoning around obstacles, moving down a hallway and counting doors, finding a wall, and aligning to a wall. Each of these is briefly described below.

Dead Reckoning Around Obstacles

The obstacle-avoiding activity implemented on Toto was a fairly uninteresting *ad hoc* algorithm. (See the next section for a more principled method of avoiding obstacles.) The activity started by simply moving the robot directly towards the target point. If the robot encountered an obstacle before reaching the target, it would invoke the going-through-doors primitive (despite the fact that there was usually no door) to get past the obstacle. It would then try again to get to the goal from its new position. This activity would fail if the going-through-door primitive failed, or if a time limit (based on the original distance to the target) was exceeded. The algorithm is *ad hoc*, but it seemed to work.

Following Hallways and Counting Doors

Door-counting was done by invoking the wall-following primitive until it failed. If the failure was due to a door, the robot dead-reckoned around the door, incremented the door count, and resumed following the wall. This activity could fail in two ways, when the dead reckoning failed, or when the wall following activity failed due to not being able to align to the wall for two contiguous meters of travel.

Aligning to Walls

The wall-following activity required that the robot be aligned within about 15 degrees to the wall before it was invoked. To correct alignment errors greater than this, a primitive essentially identical to the wall-following primitive was invoked, but using the front two sonars rather than the side sonars. This reliably aligned the robot to a wall when its initial alignment error was as large as 45 degrees. This activity failed if the alignment correction

was greater than 45 degrees, or if no alignment could be found. The alignment was verified by attempting to follow the wall for a short distance.

Finding Walls

Walls were located by invoking the dead-reckoning activity using a target located at an extreme distance in the direction where the wall was expected to be found. The activity was allowed to run until it failed, indicating the presence of a large obstacle that the robot could not get around. The robot then verified that the obstacle was actually a wall by invoking the alignment activity. The location of the wall was also compared to the expected location of the wall in an internal world model when one was available. This method worked fairly well, but it could be fooled by large flat obstacles such as couches, especially when no world model was available. It is probably not possible for a robot equipped only with sonars to distinguish between a wall and a long wooden couch in the absence of an a priori world model.

5.2.6 Experiment 1

The first experiment was a demonstration of a complex navigation task. The goal of the experiment was to move the robot from inside a cluttered office to another office on the other side of the building. (See figure 5.2-11.) For this experiment, a linear sequence of task-schema invocations was constructed by hand. The robot started at location (a). The sequence consisted of the following steps:

1. Dead reckon to a location (b) near the office door.
2. Move through the door.
3. Dead reckon to (d). (The robot was made to dead reckon rather than follow the walls at this point to test its abilities to recover from large dead-reckoning errors, as well as its ability to deal with large, unexpected obstacles.)
4. Move east[4] until encountering a wall.

---

[4]Directions are given relative to the figure, i.e. North is up, East is right, etc. The actual orientation of the building is different.

5. Align to the wall and turn left.
6. Follow the wall as far as possible.
7. Move slightly away from the wall. (This was done to keep the robot away from a low ramp which was at the end of the wall. The robot's sonars could not detect this obstacle.)
8. Move north until encountering a wall.
9. Align to the wall and turn right.
10. Follow the wall and turn left at the second door.

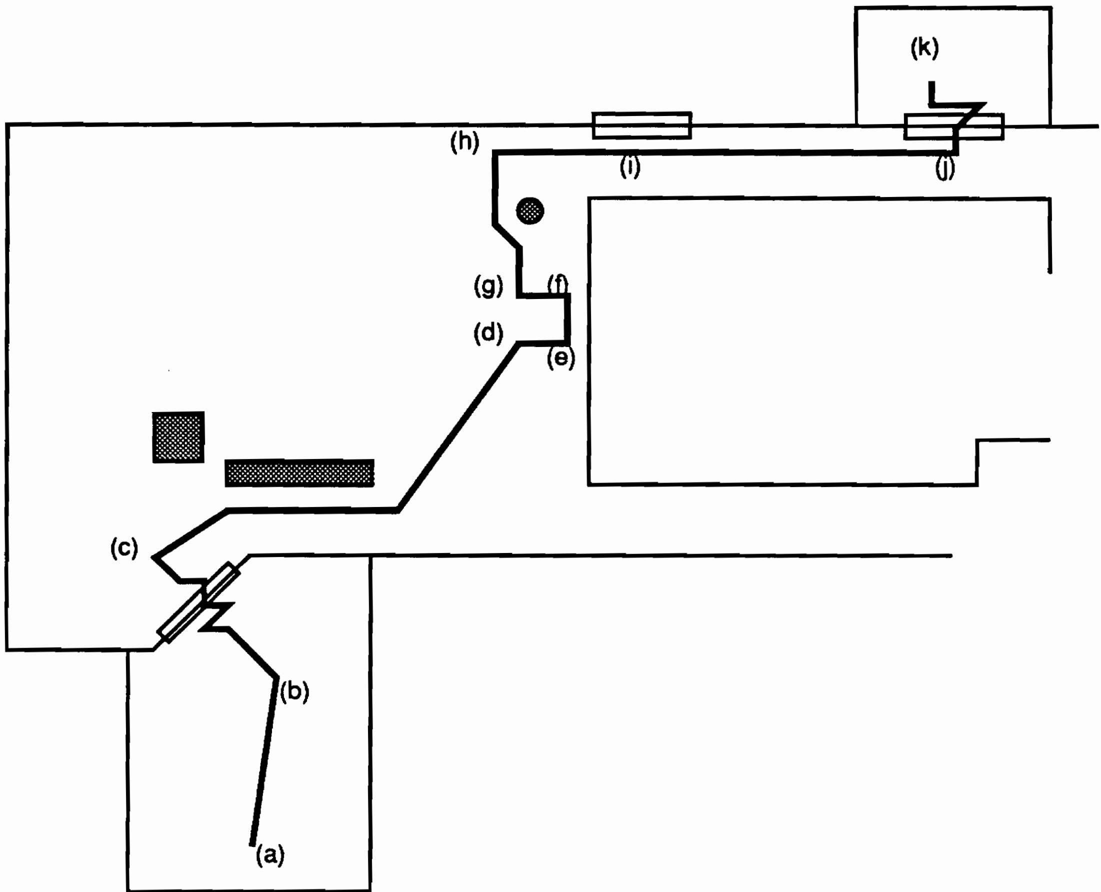


Figure 5.2-11: A complex indoor navigation task.

A diagram of one run of the experiment is shown in figure 5.2-11. The figure was generated by hand from a video tape of the run, and many details of the robot's path are omitted for clarity. The robot started out at (a) at the far end of the first office which was cluttered with desks, chairs, wires, boxes, and piles of papers (in short, a typical graduate

student office). The robot dead-reckoned to a position (b) near the door to the office. It then went through the door into the "playroom", a large lounge filled with furniture (c). It then dead-reckoned to (d) near the north-south wall on the east side of the room. Along the way it had to avoid a large couch (which the robot did not know would be there) requiring a considerable detour.

At this point, the accumulated dead-reckoning error was quite severe, on the order of a meter or two with ten to twenty degrees of orientation error. To correct this, the robot searched for and aligned to the north-south wall at (e). This allowed it to correct dead-reckoning errors in orientation and in its x position, but not in its y position.

The robot then attempted to follow the wall northward. However, this wall was made of whiteboard panels rather than the usual drywall. This made it exceptionally reflective to the robot's sonars, causing the wall-following to fail at (f). (It turned out later that there was a bug in the code which also contributed to the failure.) The robot then moved some distance away from the north-south wall to (g) and attempted to locate the far east-west wall, encountering another unexpected obstacle (in the form of the experimenter) along the way. The decision to stand in the way of the robot was made on the spur of the moment in order to demonstrate the robot's ability to distinguish between a wall and a small obstacle. When the far wall was successfully located at (h) the robot followed the wall, passing one door at (i) until it came to the door of the target office at (j). From there it went through the door to (k) which completed the task. The total length of the path is approximately 30 meters.

The experiment was formally performed four times. Two of the four runs ran to completion. The other two runs encountered failures for which no recovery procedures existed. However, at no time did the robot collide with an obstacle or get lost without realizing that something was wrong. (cf. [Thorpe90])

Discussion

This initial experiment had a number of shortcomings. The following discussion addresses these, and also discusses what lessons were learned.

This experiment was controlled by a top-level task schema which consisted of a linear sequence of task-schema invocations. This sequence was generated by hand, but it is well within the realm of current planning technology to generate such sequences automatically.

Because the top-level schema was a linear sequence it would fail if any of its constituents failed. This made it fairly fragile, failing about half the time. However, at no time did a failure ever go undetected. The robot never inexplicably turned in the wrong direction or collided with an obstacle. The robot never thought it had achieved the goal without having actually achieved it.

Finally, the last section of the experiment depended on both the target door and the door before it being open. If either of these doors were closed, the experiment would fail non-cognizantly. This issue is addressed in the next experiment.

Despite these drawbacks, the experiment does demonstrate a number of interesting things:

First, robust goal-directed navigation is possible using a real robot with crude sensors in an unmodified environment and without an accurate world model. No measurements were used to generate the plan with the exception of the position of the door between (b) and (c), and even these were only accurate to 10 cm, only slightly less than the radius of the robot.

Second, the model of a primitive activity with cognizant failure is a reasonable interface between real robot hardware and higher-level systems. Useful primitive activities can be engineered which reliably control robots under real-world conditions of noise, unpredictability and limited computation. These primitives can be easily composed by a higher-level system to perform a complex and useful task.

Third, ALFA is a convenient and useful tool for implementing complex primitive activities. Once the algorithm for wall following was derived, implementing it in ALFA was straightforward.

5.2.7 Experiment 2

One of the major shortcomings of the first experiment was the way in which the goal office was located. Simply counting doors is not a very robust way of locating a particular office. If a door is unexpectedly opened or closed the robot could easily end up in the

wrong place. In the second experiment the robot was able to locate a particular doorway along a hall by using a map. The robot would find the door regardless of which of the doors along the hall were open or closed and without knowing its starting location. If the target door was closed the robot would report that, too. The top-level task schema for this experiment, like the first, was generated by hand. Generating the plan automatically is easily within the realm of current planning technology.

At the beginning of the experiment the robot moved up and down the hall making an internal map of the doors and the distances between them. It then matched this internal map with an a priori map of all the doors on the hallway. It continued mapping until a unique match was found, at which point the robot could determine its position along the hall. It then went directly to the target door. In this way, the robot could locate the target door even if it was closed, and even if a door which had been open during the mapping phase was closed during the second phase, or vice versa. All of these capabilities were reliably demonstrated by the robot many times.

Example Run

It is difficult to draw a picture of this experiment, as all it would show is the robot moving up and down the hall. Instead, I will present a portion of the transcript from an actual run. This will also serve to illustrate how the system works overall. In the transcript that follows, reference will be made to the robot's "state". This indicates the position of the robot when a primitive was terminated.

This experiment assumes that the robot starts out aligned to a wall and ready to follow it. (This assumption will be discharged by the next experiment.) The robot starts mapping by following the wall until it finds a door. This occurs without mishap:

```
-Invoking primitive TOTO-FOLLOW-WALL...completed.
New state is: #{TOTO state: X: 3.79m  Y: 5.87m  Theta:317.0 degrees}
Found a door after travelling 7.541800071149057E-2 meters.
```

Now the robot must dead-reckon past the door and realign itself to the wall. Again, this happens without incident. Only a single primitive is actually invoked during the procedure, one which moves the robot forward about a meter.

-Invoking procedure TOTO-ROBUST-GO-FORWARD.
--Invoking procedure TOTO-ROBUST-GO-TO.
---Invoking procedure TOTO-GO-TO.
----Invoking procedure TOTO-TURN-TO.
----Procedure TOTO-TURN-TO completed.
----Invoking primitive TOTO-GO-FORWARD...completed.
New state is: #{TOTO state: X: 4.52m Y: 5.18m Theta:317.0 degrees}
---Procedure TOTO-GO-TO completed.
--Procedure TOTO-ROBUST-GO-TO completed.
--Invoking procedure TOTO-TURN-TO.
--Procedure TOTO-TURN-TO completed.
-Procedure TOTO-ROBUST-GO-FORWARD completed.

The preceding is what happens under ideal circumstances. However, many things can go wrong when moving past a door. The most common problem occurs when dead-reckoning errors cause the robot to drift towards the opposing door jamb so that it appears to be an obstacle. The following is a typical example of recovering in this case.

-Invoking primitive TOTO-FOLLOW-WALL...completed.
New state is: #{TOTO state: X:11.70m Y:-1.50m Theta:317.0 degrees}
Found a door after travelling 0.1490572749911064 meters.
-Invoking procedure TOTO-ROBUST-GO-FORWARD.
--Invoking procedure TOTO-ROBUST-GO-TO.
---Invoking procedure TOTO-GO-TO.
----Invoking procedure TOTO-TURN-TO.
----Procedure TOTO-TURN-TO completed.
----Invoking primitive TOTO-GO-FORWARD...FAILED!
New state is: #{TOTO state: X:11.75m Y:-1.55m Theta:317.0 degrees}

The TOTO-GO-FORWARD primitive activity has failed because it encountered an obstacle. This failure gets propagated to the TOTO-GO-TO task which initiated it.

---Procedure TOTO-GO-TO FAILED!

Fortunately, TOTO-GO-TO has a recovery procedure. It begins by invoking the door-traversal activity. This activity is usually used to wriggle *through* doors, but it can also be used to wriggle *past* them if the robot starts out headed parallel to the door.

---Invoking procedure TOTO-GO-THROUGH-DOOR.
----Invoking procedure TOTO-GO-FORWARD-AT-LEAST.
-----Invoking primitive TOTO-GO-FORWARD...FAILED!
New state is: #{TOTO state: X:11.95m Y:-1.73m Theta:317.0 degrees}
----Procedure TOTO-GO-FORWARD-AT-LEAST FAILED!
----Invoking procedure TOTO-ROBUST-TURN-TO-CLEAR-HEADING.
-----Invoking procedure TOTO-GO-FORWARD-AT-LEAST.
------Invoking primitive TOTO-GO-FORWARD...completed.
New state is: #{TOTO state: X:12.02m Y:-1.80m Theta:317.0 degrees}
-----Procedure TOTO-GO-FORWARD-AT-LEAST completed.
----Procedure TOTO-ROBUST-TURN-TO-CLEAR-HEADING completed.
----Invoking procedure TOTO-GO-FORWARD-AT-LEAST.
-----Invoking primitive TOTO-GO-FORWARD...completed.
New state is: #{TOTO state: X:12.76m Y:-2.49m Theta:317.0 degrees}
----Procedure TOTO-GO-FORWARD-AT-LEAST completed.
---Procedure TOTO-GO-THROUGH-DOOR completed.

Now the robot is clear of the door jamb, but it is no longer next to the wall. The robot therefore tries again to go to a point along the wall past the door.

---Invoking procedure TOTO-ROBUST-GO-TO.

----Invoking procedure TOTO-GO-TO.
-----Invoking procedure TOTO-TURN-TO.
------Invoking primitive TOTO-TURN...completed.
New state is: #{TOTO state: X:12.76m Y:-2.49m Theta:137.0 degrees}
-----Procedure TOTO-TURN-TO completed.
-----Invoking primitive TOTO-GO-FORWARD...completed.
New state is: #{TOTO state: X:12.43m Y:-2.18m Theta:137.0 degrees}
----Procedure TOTO-GO-TO completed.
---Procedure TOTO-ROBUST-GO-TO completed.
--Procedure TOTO-ROBUST-GO-TO completed.

This time, all goes well, and the last step is to turn the robot in preparation for following the wall again.

--Invoking procedure TOTO-TURN-TO.
---Invoking primitive TOTO-TURN...completed.
New state is: #{TOTO state: X:12.43m Y:-2.18m Theta:317.0 degrees}
--Procedure TOTO-TURN-TO completed.
-Procedure TOTO-ROBUST-GO-FORWARD completed.

The robot now continues mapping by following the wall again. The robot continues mapping until it finds an unambiguous match between the map it is generating and the *a priori* map it was given. At that point, the robot knows its location along the hallway, and can proceed directly to the target door.

5.2.8 Experiment 3

The final experiment tested the ability of the robot to recover from large errors in its position. For this experiment, the robot was equipped with a magnetic compass which provided a crude measure of the robot's orientation. Because of local magnetic anomalies (probably caused by the steel beams in the building), the compass could be off by as much as 40 degrees.

In order to fix its position, the robot first aligned itself to the compass. Using this crude alignment it was able to locate and align itself to one of the walls. Finally, the robot aligned itself to the second wall. To test the accuracy of the alignment, the robot then dead-reckoned to a position two meters away from the corner. This final position was reproducible to within a few centimeters, a distance comparable to the resolution of the sonars, despite the fact that the robot's initial position and orientation were essentially unknown.

A diagram of this experiment is shown in figure 5.2-12. The area mapped is the lower-left corner of figure 5.2-11. The robot was started at various locations in the shaded area (1). It would align itself to the compass, and head towards the left wall (2). When it reached the wall, it would align itself and head towards the southwest corner (3). At this point, the robot has corrected large errors in all three of is degrees of freedom. (Compare this method of fixing the robot's position with the analytical method presented in [Miller84].)

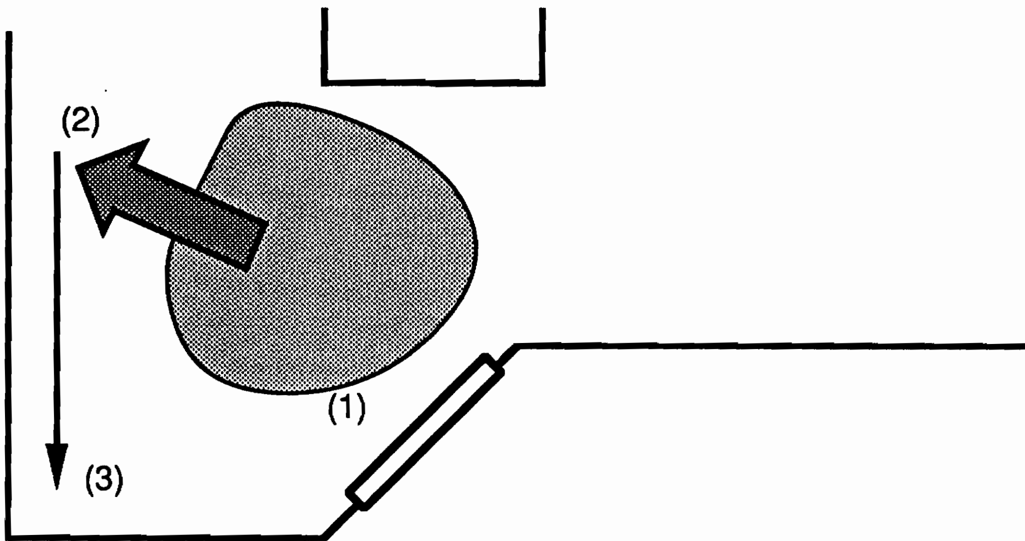


Figure 5.2-12: Reducing positional uncertainty by aligning to walls.

For truly robust behavior, the robot really needs the ability to locate itself starting at any location in the building. This is because dead reckoning errors can potentially, through repeated alignment failures, cause the robot to become hopelessly lost. To accomplish this, however, the robot needs some sort of sensor by which it can unambiguously determine its

position at least to the resolution where some other sensor modality can position the robot more accurately. Toto has no such sensors. There is simply no way to tell one corner apart from another using sonars alone. Matching an internal map to an *a priori* map to determine position will work only if the robot can construct a large enough map without interim failures (e.g., [Miller91]). A better solution would be to equip the robot with some extra sensors which would allow it to determine, say, when it was in the playroom (e.g., [Braunegg90], [Sarachik89]). Then when the robot got lost it could recover by wandering randomly until it got to the playroom, locate a corner of the playroom to realign itself, and then proceed with its initial task.

5.3 Robbie

The preceding experiments demonstrate the viability of the control layer and its ability to interface to the sequencing layer. However, it leaves a great many issues unaddressed. To remedy this, a more complete version of ATLANTIS was implemented on Robbie, one of the Jet Propulsion Laboratory's planetary rover testbeds.

Robbie is a large six-wheeled robot, three meters long by one and a half wide by two high. The body is divided into several segments connected by hinges and pivots, allowing the robot to conform to extremely rough terrain. It weighs over a ton, and moves at a top speed of 2 meters per minute with a minimum turning radius of about three meters. Robbie's main sensor is a pair of stereo cameras mounted on a pan-tilt head. The cameras have a field of view of about 30 degrees by 23 degrees, and can be aimed within a 180 degree arc in front of the vehicle. Robbie cannot see to the rear, nor can it see anything closer than about 2 meters in front of the vehicle. Robbie also has a set of joint encoders and inclinometers. These are useful for detecting certain sorts of problems such as driving over large rocks or into holes. Robbie also has an inertial gyro used to fix the robot's orientation with respect to an absolute coordinate frame. This gives the vehicle some fairly good dead-reckoning abilities, though the error can still be quite significant.

Robbie has two VME card cages, each with a 68020 microprocessor with several megabytes of RAM. One of the card cages is dedicated to controlling the robot's six drive motors and the pan-tilt head. The other contains several specialized image-processing boards which are used to support stereo vision calculations. Robbie can process a stereo vision frame in about twenty seconds [Matthies91]. This card cage also runs T [Slade87],

a dialect of Lisp, on its 68020. It is on this processor that the software described in this section was implemented.

Robbie is by far the largest and most sophisticated robot of the ones used in this research. It is also the least robust for two reasons. First, the software infrastructure on the robot is such that implementing a complete ATLANTIS control layer would be exceedingly difficult. The existing motor control software is deeply ingrained. To get around this, an interface was constructed on top of the existing control software to make it look as if it were an ALFA program. Thus, the interface to the control layer conformed to the ATLANTIS architecture, even though its functionality did not conform completely.

Second, the robot does not have the sensors required to implement a reasonable control layer even if the infrastructure supported it. The joint encoders and inclinometers can detect an obstacle only when the robot is already on top of it. At that point the only thing the robot can do is to back up along its previous path in hopes of extricating itself. The only sensor capable of detecting obstacles remotely is the stereo cameras and the processing required to extract useful information from them takes on the order of a minute per frame. Thus, the hardware simply does not support the sort of sophisticated reactive control which was implemented on Toto.

Instead, Robbie was used to test the interaction of the ATLANTIS sequencing layer and the deliberative layer to coordinate two physical processes, controlling the robot's heading and the pan-tilt head, and two computational processes, the generation of path plans and the processing of vision data. Each of these is discussed in turn in the following sections.

5.3.1 Navigation

The navigation algorithm was based on Navigation Templates (NaTs) [Slack90]. Navigation Templates provide a method for generating a preferred direction of travel at a point from a high-level symbolic description of the goal location and the surrounding obstacles.

There are two kinds of NaTs, substrate Nats (s-NaTs) which are associated with goals, and modifier NaTs (m-NaTs) which are associated with obstacles. Goals in the NaT system are not necessarily locations. The robot may have the goal of tracking a curving

trajectory, or simply to travel in a particular direction. Thus, s-NaTs are generalized vector fields which describe the preferred direction of travel at any given point assuming no obstacles are present.

The heart of the NaT system is an algorithm which combines the nominal direction of travel given by the goal s-NaT with constraints placed upon the motion of the robot by obstacles described by m-NaTs. Every m-NaT is assigned a *spin* which indicates whether the robot should avoid the obstacle by passing it on the left or on the right. A clockwise spin means the robot should avoid the obstacle by passed to the left of the obstacle; a counterclockwise spin means the robot should pass to the right. The algorithm combines all the constraints placed by all the m-NaTs to generate a preferred direction of motion at the current location. This direction is called the *gradient*, an unfortunate choice of terms because the vector field defined by this algorithm is not necessarily a gradient field (i.e. the gradient of a scalar potential field). However, the term is deeply ingrained in the literature, and so I will retain it.

NaTs have two significant advantages over other methods of mobile robot trajectory generation. First, the algorithm is fast, so the robot's trajectory can be updated often resulting in fewer accumulated errors. Second, the algorithm is immune to local minima problems characteristic of other vector-field-based algorithms [Slack90].

Unfortunately, the algorithm has two serious problems as well. First, it assumes that the m-NaT spins are correctly assigned by an oracle. Second, it does not take into account non-holonomic constraints [Barraquand89]. This can lead to serious problems when the vector field generated by the algorithm requires the robot to make a sharp turn. (See figure 5.3-1.)

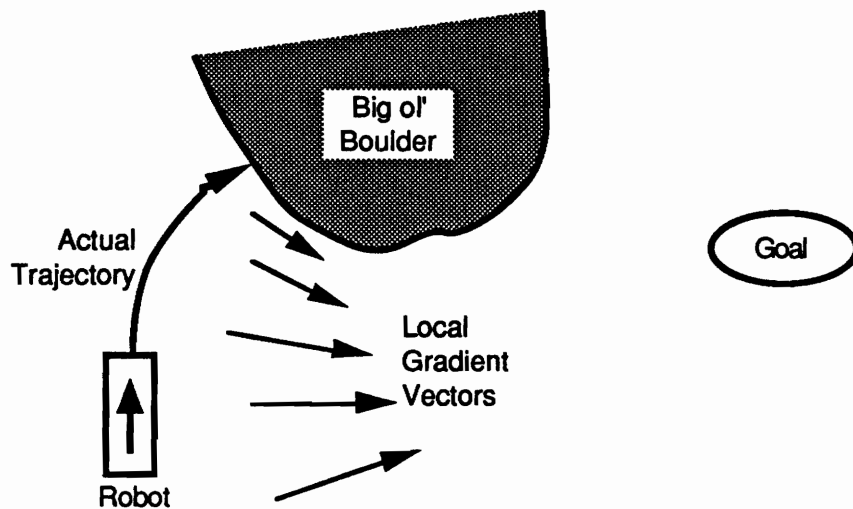


Figure 5.3-1: Why non-holonomic constraints must not be ignored.

Both of these problems were addressed together by incorporating non-holonomic constraints into the spin-assignment algorithm. Non-holonomic constraints are significant only when an obstacle intersects the robot's minimum radius turn. (See figure 5.3-2). In such a case, the robot can get around the obstacle only one way without backing up. Thus, by constraining the spin of that obstacle to move the robot in the opposite direction, problems with non-holonomic constraints can be avoided in most cases. (This approach does have problems, however. See section 5.4.3.2 for a description and a solution.)

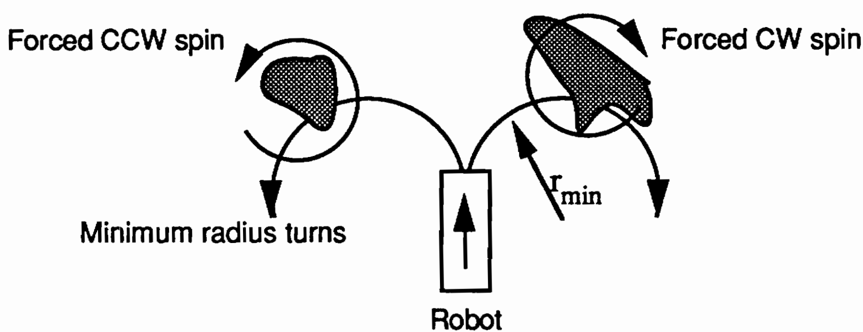


Figure 5.3-2: Satisfying non-holonomic constraints through proper spin assignment.

For obstacles whose spins are not determined this way, a heuristic method is used based on whether the obstacle is to the left or to the right of the robot's current heading,

and whether it is to the left or right of the current s-NaT direction. These two influences are weighted according to the distance of the obstacle from the robot (closer obstacles cause the robot direction to override the gradient direction) and are then averaged.

A process which periodically computes the gradient and updates the robot's command queue accordingly essentially provides an activity which moves the robot towards the goal and around obstacles. This activity requires support activities to gather and process vision data in order to provide the gradient algorithm with the locations of obstacles. Also, this activity can fail in several ways. First, it is possible that an obstacle will be detected which intersects both minimum-radius trajectories. Second, the robot may simply collide with something due to actuator error, slippage, or sensor error. Finally, the heuristic spin assignment algorithm occasionally makes a poor choice and sends the robot along a path which does not lead to the goal. This last failure is particularly serious because it can be very difficult to detect and recover from.

These issues are addressed in subsequent sections.

5.3.2 Pan-tilt Control

Robbie is equipped with state-of-the-art vision processing boards and an extremely fast stereo matching algorithm [Matthies91]. Nevertheless, processing vision data still takes a significant amount of time. Thus, it is important that the robot direct its cameras selectively towards areas where it needs data [Miller86], [Dean90b]. It also needs to keep track of where it has looked and where it has not so that it does not wander blindly into unscanned areas.

The camera-pointing algorithm which was used worked as follows. If there were any unscanned areas between the robot's current heading and the current gradient direction it would look there first, preferring to look closer to its own heading than towards the gradient. If there were no unscanned areas, then the robot would scan the area into which it was currently heading -- straight ahead if it was moving straight, slightly to one side if it was turning. This provided many redundant scans of the robot's trajectory to insure that it would not collide with obstacle missed by one scan.

5.3.3 Path Planning

A simple path planner was implemented which did a heuristic grid-based search for a clear path to a goal based on the robot's current knowledge of the configuration of the world. The output of this planner was a vector field which was used as the substrate NaT for the navigation activity. The planner handled multiple goals by choosing the goal which was easiest to achieve. No goal-ordering optimization was done.

A second planner was implemented to compute recovery paths for when the gradient-following activity failed. Again, a simple search algorithm searching a discretization of the robot's configuration space was used [Barraquand89]. This planner was invoked whenever the normal NaT-based navigation activity failed.

The most interesting thing about these planners, as with the sequencing layer on Toto, is how uninteresting they are. Both are essentially toys, yet in conjunction with the rest of the system were able to produce quite robust behavior.

5.3.4 Vision Processing

Vision processing occurred in two stages. First, the stereo images were correlated to produce a depth map. The resulting depth map was then sent through a coordinate transformation to turn it into a height map. The second stage took the height map and extracted obstacle data from it using a simple thresholding algorithm. Nearby points were then grouped together by a cluster-analysis algorithm to form "boulders" which were then used by the gradient algorithm and the path planner [Slack90].

5.3.5 The Sequencing Layer and the Deliberative Layer

The deliberative layer was responsible for performing the path planning and vision processing computations. The sequencing layer coordinated these computations with the physical activities of navigation and camera aiming.

Because the sequencing and the deliberative computations had to take place within a single Lisp process using a system which provided no multitasking capabilities, the deliberative computations had to be interleaved with the sequencing computations. This was done by structuring the deliberative computations as engines [Dybvig89]. An engine

is a continuation which may be invoked for a fixed period of real time. If the computation is not finished by the time limit, the engine returns a new continuation which, when called, resumes the computation from the point it left off, again for a specified amount of time. This process repeats until the computation is complete.

The sorts of interactions which occurred among these activities are best illustrated by an actual run, which is described in the next section.

5.3.6 The Experiment

The experiment was conducted in a dry creekbed next to the California Institute of Technology Jet Propulsion Laboratory. The course contained a variety of obstacles, including shrubbery, and all sizes of rocks from pebbles to boulders. A rough diagram of the course is shown in figure 5.3-3. (This figure is reconstructed from actual data taken during the experiment, and therefore shows the locations of obstacles as they were perceived by the robot's vision system. As of this writing, there is no way to reproduce the actual terrain in this document.)

Figure 5.3-3: The initial configuration for the outdoor navigation experiment.

The robot was given three goals, one located 15 meters ahead of the rover, and two located 30 meters away at 45 degree angles. It was to visit all three, but in no particular order. The robot was not given any advance knowledge of the state of the world except that the area immediately in front of it (shown by the light polygon) was clear of obstacles. This was necessary because the robot has no way to scan the area immediately in front if itself, so it must be told *a priori* that this area is clear or it cannot begin to move.

The robot began by moving towards goal B because it was the closest. Once goal B was reached, the robot chose to go to goal A because the obstacles which it discovered to its left blocked the direct route to goal C. (See figure 5.3-4). The areas covered by the vision scan are shown as light polygons. The obstacles detected by the robot are shown as hollow circles concentric with the obstacle.

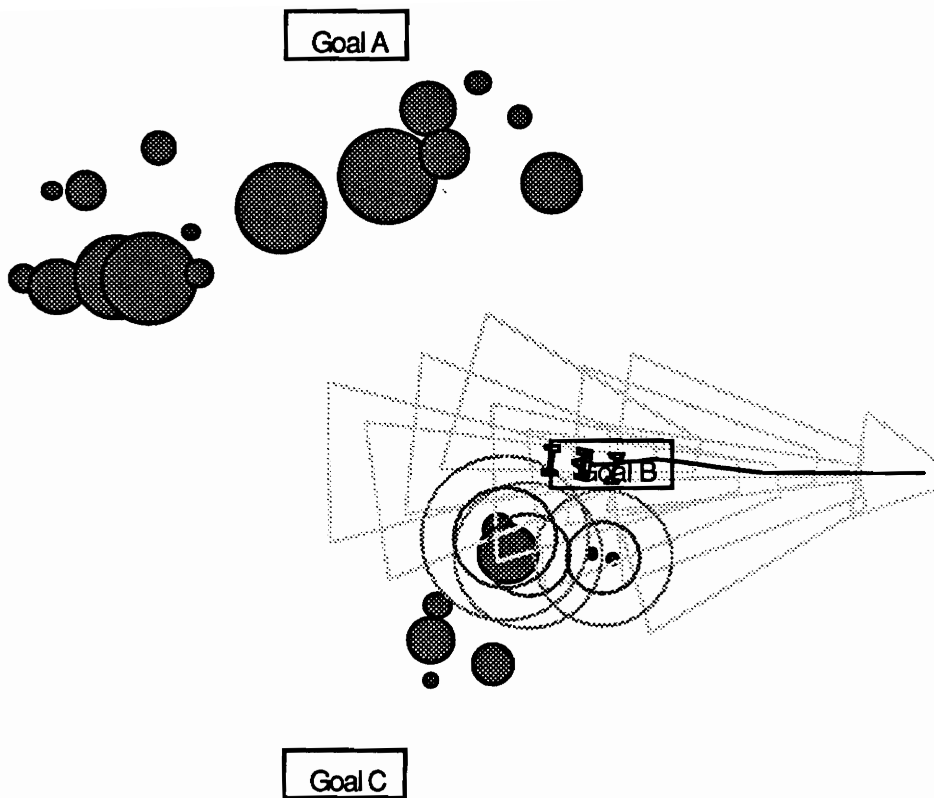


Figure 5.3-4: Goal B has been achieved.

On the way to goal A many obstacles were discovered which caused the path planner, which had been running asynchronously in the deliberative layer, to conclude that goal C would actually be easier to achieve than goal A. Goal A was therefore temporarily abandoned in favor of goal C. (See figure 5.3-5.) The robot would have returned to goal A eventually, but time did not permit the experiment to run to completion. The total length of the path is about 30 meters, and was covered in about a half an hour. The robot was moving continuously throughout the experiment except when it had to make sharp turns into areas it had not yet scanned. Then the robot was forced to wait for the vision processing to complete. However, all the planning occurred while the robot was moving.

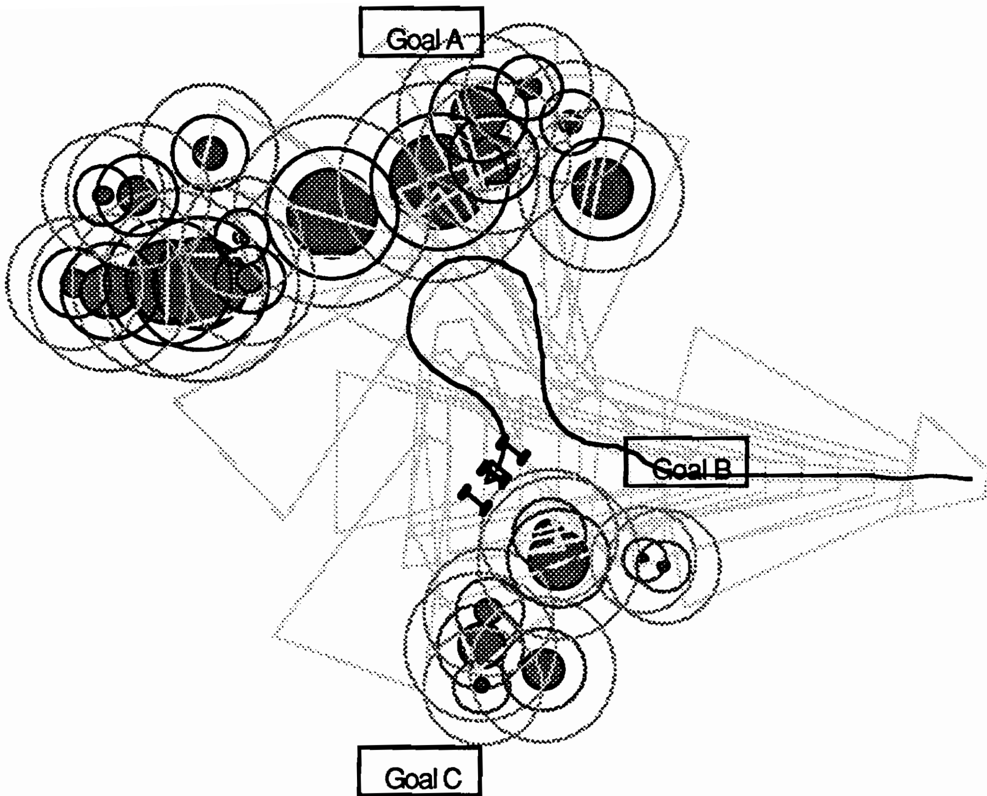


Figure 5.3-5: Goal A has been temporarily abandoned in favor of goal C.

This experiment illustrates that ATLANTIS[5] can produce robust navigation behavior on a real robot. It also shows that the ATLANTIS sequencing layer and the deliberative layer can work together to produce goal-directed behavior based on the results of deliberative computations performed asynchronously.

There are many factors which are difficult to control on a real robot. Also, experiments on Robbie are expensive and time consuming. Therefore, a number of experiments were run on a Robbie simulator so that the system could be tested on a larger variety of situations and monitored in more detail. These experiments are described in the next section.

---

[5]All Tight Limits Are No Trouble; It Survives

5.4 Simulator Experiments

Great care must be used when employing a simulator. It is all too easy to simulate away significant problems. When one is making claims about an architecture which is supposed to be robust in the face of real-world errors, the use of a simulator is suboptimal at best. However, simulating an actual rather than a hypothetical robot, and then verifying the simulated results on the real robot can go a long way towards lending credence to simulator results. The main reason a simulator was used is that there is a tremendous overhead involved in using Robbie. Setting up an experiment on the real robot required an entire day, usually with a week of advance notice. One can get much higher experimental throughput on a simulator, and also experiment with variations of the system parameters which would be impossible in reality.

The simulator used a fairly high-fidelity model of Robbie's sensors and kinematics. The two main differences between the simulator environment and the real robot is that the world in the simulator consisted of polygonal obstacles, and the simulated vision tended to work better than the real thing. The simulated vision did include noise, but the sorts of errors encountered in the real world turned out to be quite difficult to model. The simulation ran in real time, which was something of a handicap because the computer used in the simulation runs was less powerful than that on the real robot. Nevertheless, ATLANTIS was able to control the simulated robot with little difficulty. Most of the code used with the simulator was the exact same as that used on the real robot.

5.4.1 Experiment 1: Replication of Arroyo Results

The first experiment was an attempt to replicate the experiment conducted on the actual robot. (See figure 5.4-1.) The obstacles detected by the real robot were used as the simulator's world model. Note that many of the obstacles in the model went undetected in the simulator run. (This is indicative of the situation on the real robot where many small obstacles were not detected by the vision system; there was simply no way to gather data on their location so that they could be displayed in the figures in the previous section.) The less loopy path in the simulator run is a reflection of the better "vision data" provided by the simulator. The robot saw the obstacles from further away, and so it was able to take corrective action sooner.

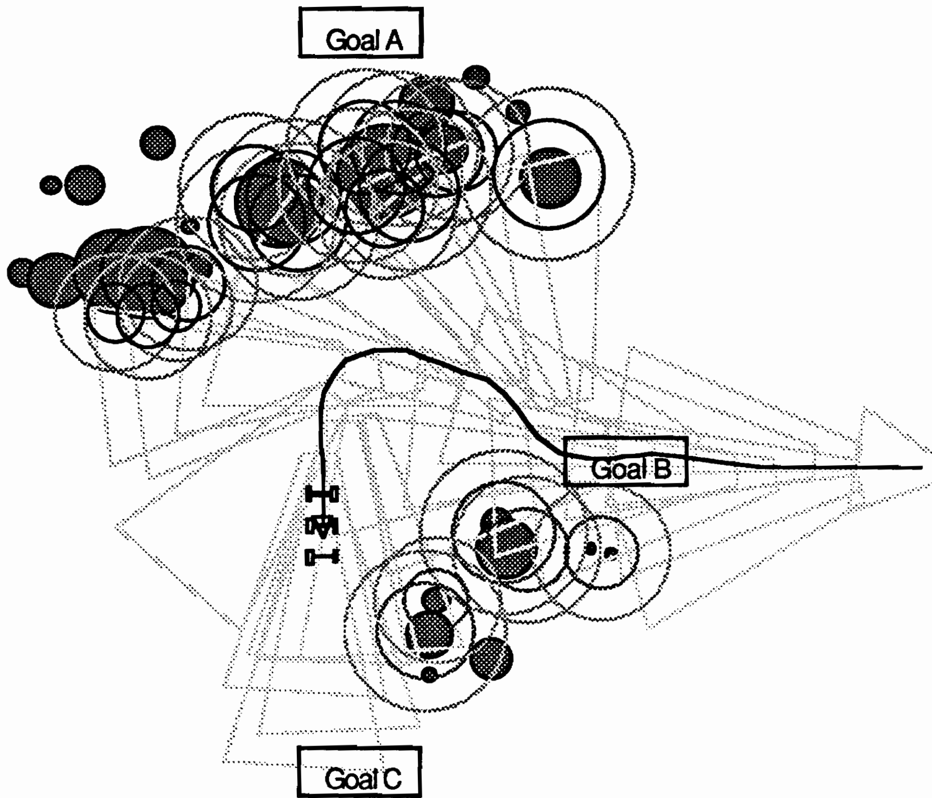


Figure 5.4-1: A simulated recreation of the outdoor navigation experiment.

This experiment demonstrates that the behavior of the simulator is not totally out of step with the performance of the actual robot.

5.4.2 Experiment 2: Complex Navigation

In order to test the robot on more challenging problems, a cluttered artificial world full of obstacles was randomly generated. (See figure 5.4-2.) This world was the basis for a variety of experiments which demonstrate various aspects of the system's performance. The cluttered area measures about 50 meters wide by 40 high.

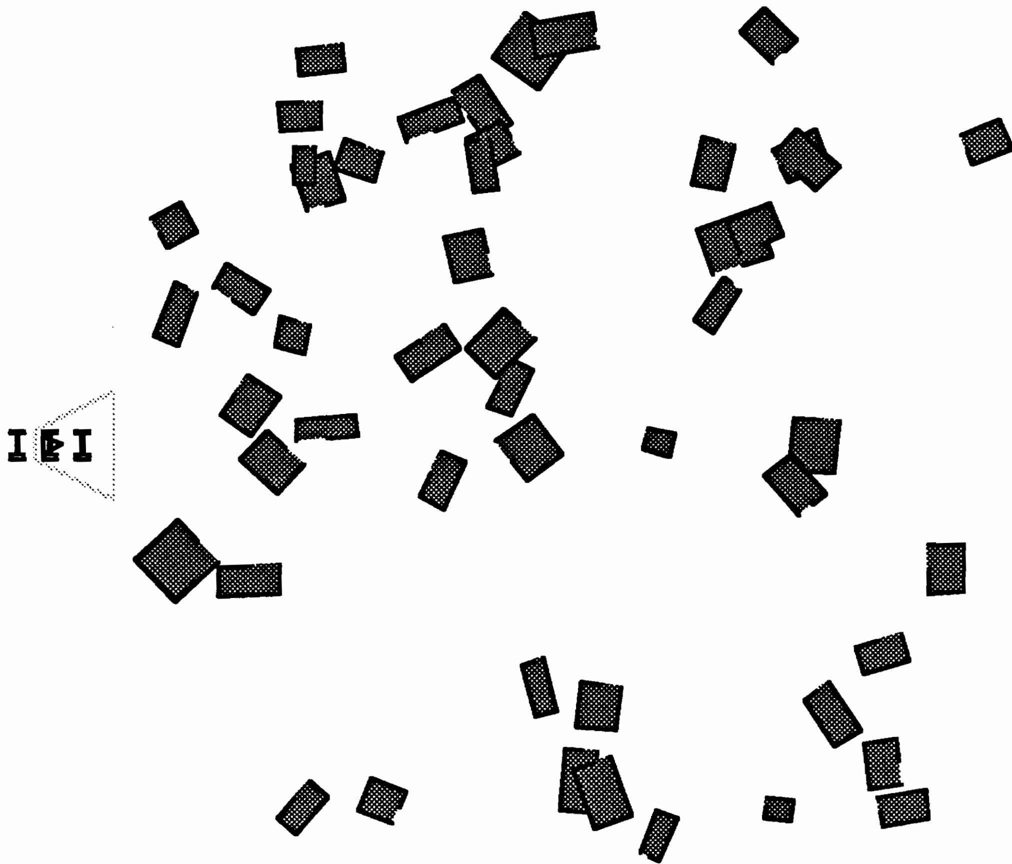


Figure 5.4-2: The simulator world.

Many tests were run on this terrain. Figure 5.4-3 shows a typical example. The setup is similar to the experiment run on the real rover. There were three goals. The robot temporarily abandoned one goal (B) in favor of another (C) when the direct route to goal B was found to be blocked.

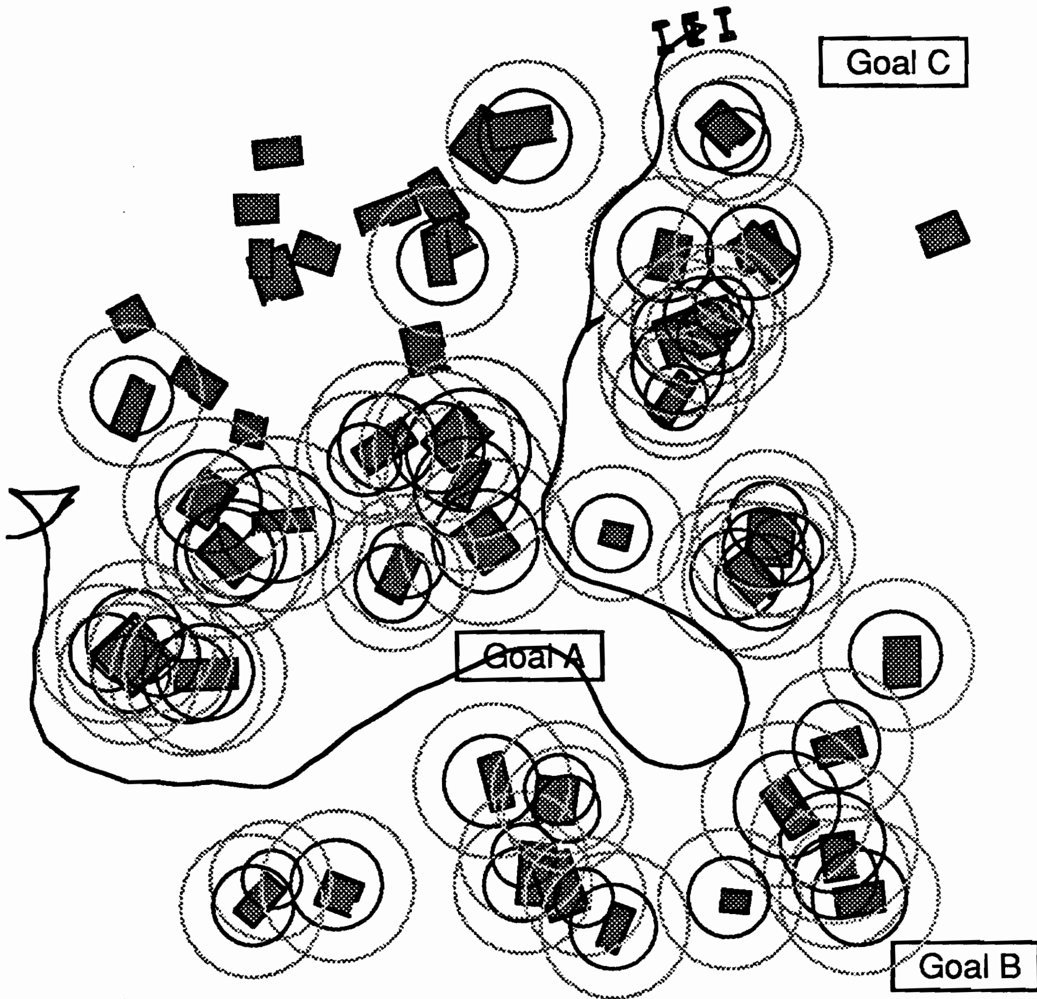


Figure 5.4-3: A typical simulator run.

One interesting feature of this example is that in several instances the robot had to back up to extricate itself from tight situations that it did not see until it got too close. At the beginning of the run (at the left side of the figure) it started to turn towards the north. When that route turned out to be blocked, the robot had to back up and turn south instead. Another example of this can be seen halfway between goal A and the robot's position in the figure. These backup maneuvers were performed when the robot found itself having to force the spin of a boulder to be both clockwise and counterclockwise at the same time. In such cases, the robot backed up. (See section 5.4.3.2 for a more principled way of generating backup maneuvers.)

5.4.3 Experiment 3: Rocks and Martians

The third experiment tested the ability of the system to control a composite activity more complex than just navigation. In this experiment, the robot was given three tasks which might be faced by an autonomous rover operating on the surface of Mars: delivering rock samples, keeping itself fueled, and photographing martians.

5.4.3.1 Martian Physics

To support these tasks, the physics of the simulator was augmented. Rock quarries were added which produce rocks of various colors at random intervals. The rover is informed of how many rocks are available at any given time. Orders for rock samples could be received at any time. An order specifies what color rocks are required, how many are needed, and where they are to be delivered. The storage space available on the rover is limited, so some orders might require more than one trip. On the other hand, there is enough storage to allow small orders to be combined.

A home base was set up where the rover could refuel. Fuel consumption was made to be a function of real time rather than distance travelled. This is an accurate reflection of the physics of Robbie which is powered by a motor generator and consumes roughly the same amount of fuel whether or not it is moving. A real Mars rover presumably would use more fuel (or battery power) while moving than while standing still. However, even a stationary rover needs power to run its computers. Making fuel consumption a function of time causes idle computation to have a significant cost, a situation that accurately reflects reality. The fuel capacity of the robot is fixed.

The martians that roam the simulator are simple, Newtonian creatures. They move in straight lines at constant velocities, bounce elastically of the boundaries of the world, and occasionally change speed and direction. The locations of all the martians in the world are known to the rover. (Perhaps there is an imaging orbiter tracking the elusive creatures.) Because martians move, the opportunity to photograph one must be seized when it arises or the martian will be gone. The robot can plan a path to rendezvous with a martian, but this plan may fail if the martian changes direction or if unexpected obstacles are encountered.

All of these tasks interact with each other. An opportunity to chase a martian may cause the robot to temporarily abandon a delivery task. When the martian has been

successfully photographed the robot may have moved enough that resuming the original delivery task no longer makes sense. And, of course, the robot must never let its fuel supply run so low that it cannot return to its home base.

To guide the robot, a task planner was constructed which chooses the task that the robot should work on. The task planner is a linear planner which explores total orderings on tasks [Miller85]. The output of the task planner is the location where the robot should go given the current situation, and what the robot should expect to be able to do once it gets there: collect or deliver a rock sample, photograph a martian, or refuel. This planner runs periodically (in the deliberative layer, of course) and makes sure that the robot is always pursuing a reasonable course of action.

Thus, in this experiment the following processes were running at various times in the deliberative layer:

- Task planning
- Computing martian intercept trajectories
- The NaT gradient algorithm
- Vision processing
- Global path planning (i.e. spin assignment)

There were two physical processes: moving the robot and aiming the camera. The collecting and delivering of samples was modelled as discrete actions, although the robot did have to be stopped for them to succeed. (See Chapter 6 for some speculations about using ATLANTIS to do manipulation.)

5.4.3.2 Algorithm Changes

A number of modifications were made to the underlying algorithms and simulator parameters for these experiments. First, more noise was introduced into the simulated vision data. This was done in three ways. First, the number of simulated stereo vision points generated each cycle was reduced by about a factor of 10 (from 2000 points per cycle to 300). Second, the amount of random noise included in each point was increased. Third, the data was filtered by a "myopia" factor which deleted a certain number of points

based on the distance from the robot. The further a point was from the robot (at the time the frame was grabbed) the more chance it had of being deleted.

The upshot of these changes was that the simulated vision was now very unreliable. Boulders that were far away had a high probability of not being seen at all. Small boulders also had a high probability of being overlooked, even if they were close. In addition, the random noise occasionally produced "phantom boulders" in places that were, in fact, clear.

On the positive side, the time required to process vision points was now greatly reduced. To offset this advantage, the nominal speed of the robot was increased by a factor of three, from two meters a minute to six.

These changes were made for two reasons. First, the additional sensor noise made it very important for the robot to manage its sensory processing correctly. By scanning an area multiple times the robot could get a very good idea of what was really out there, but this had to be balanced against the need for timely responses. Second, the high amount of noise caused the robot to occasionally overlook and thus collide with boulders even in areas that it had scanned multiple times. This provided a way to test the collision recovery procedures. Third, the simulation now ran three times faster than before.

To support this faster-than-real-time system, the NaT gradient algorithm was reimplemented and the spin-assignment algorithm was changed. The forced-spin idea proposed in section 5.3.1 works reasonably well for small obstacles, but when obstacles are large, or when many small obstacles group together to form a large composite obstacle, the forced spin can lead to very inefficient paths. Therefore, all the NaTs were assigned their nominal spins. To avoid problems due to non-holonomic constraints, a tactical planner was written which projected the robot's forward path for a short distance to see if it was in danger of a collision due to a non-holonomic constraint. If it was, the planner would generate an alignment maneuver to bring the robot in line with the current gradient by backing up. This alignment maneuver was also checked for potential collisions. If any were found, the tactical planner would generate a skid turn. Skid turns are necessary only in extremely tight spots. The tactical planner was run at regular intervals in the deliberative layer. The grid-based strategic planner was also abandoned in favor of one which projected the gradient forward to the goal and suggested spin changes to shorten the global path.

5.4.3.3 Results

It is difficult to assess the performance of a system like ATLANTIS because there are so many factors which affect it. Among these are:

- The goals the system is given, and when they appear (i.e. when orders for rock samples are received, when a martian can be chased and photographed, and when rock samples are available).
- The number and configuration of obstacles.
- The amount of sensor noise, and how it manifests itself on any given run.
- The amount of fuel the robot can carry.
- The speed at which the robot moves.
- The amount of available memory.

It is even difficult to compare the performance of ATLANTIS to other existing systems because no other system does what ATLANTIS does, namely, pursue multiple, dynamic goals in real time in an unpredictable environment with noisy sensors and actuators. In fact, because of the random elements in the system, it is even difficult to compare the performance of ALANTIS to itself.

In the face of these difficulties, the only sort of evidence which can be offered is large numbers of actual runs which demonstrate the performance of the system. To date, the system has been run on over twenty randomly generated problems, some of which required over eight hours to complete. The total runtime logged is over thirty hours of real time. The total distance travelled over these runs is over 30 kilometers. A detailed, annotated transcript of an actual run appears in the appendix.

To give a flavor of the sort of behavior that the system produces, part of a run is presented here. The initial situation is given in figure 5.4-4. The M-shaped figures are martians. The locations of rocks of various colors are indicated in text. The robot is at the home base in the upper left hand corner. The robot's tasks are to pick up one sample of each color rock and return them to the home base.

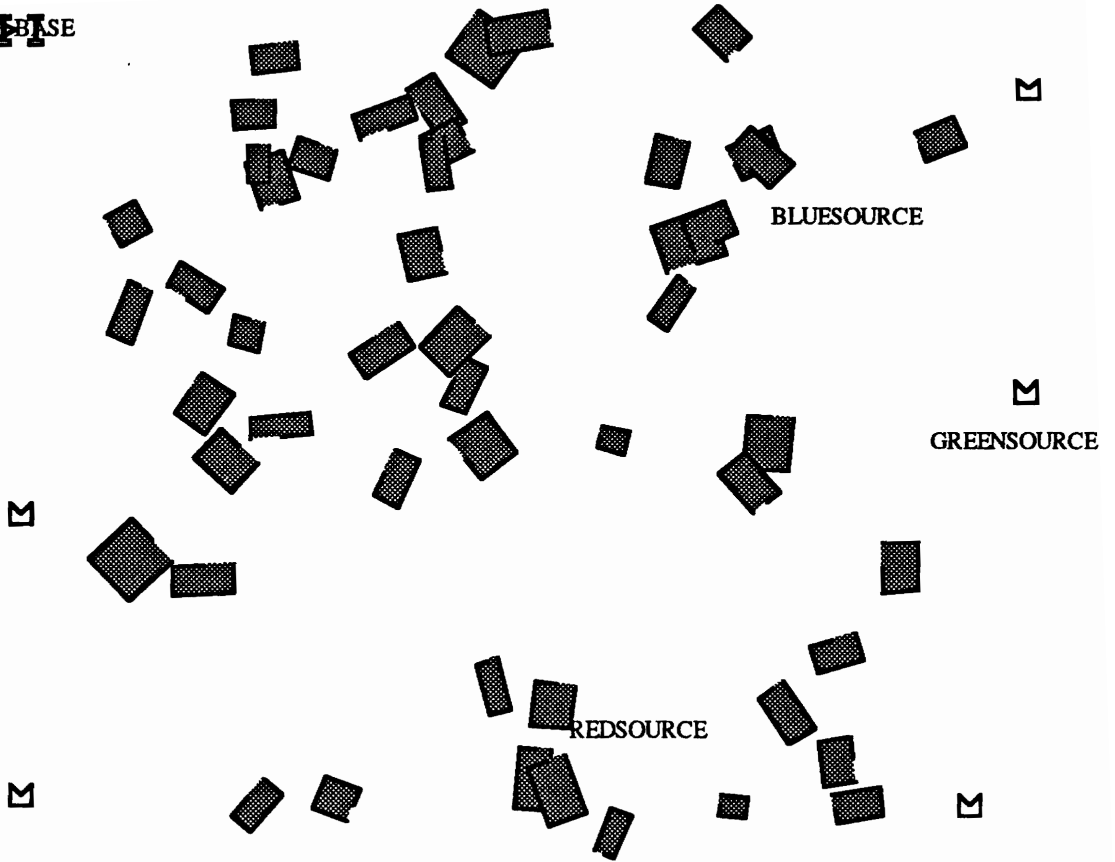


Figure 5.4-4: The start of a rock-collecting run. The M-shaped figures are martians.

At the beginning of the run, the upper martian on the left is moving towards the top of the screen. The robot starts out by planning an intercept course with this martian and getting a photograph of it. (See figure 5.4-5.)

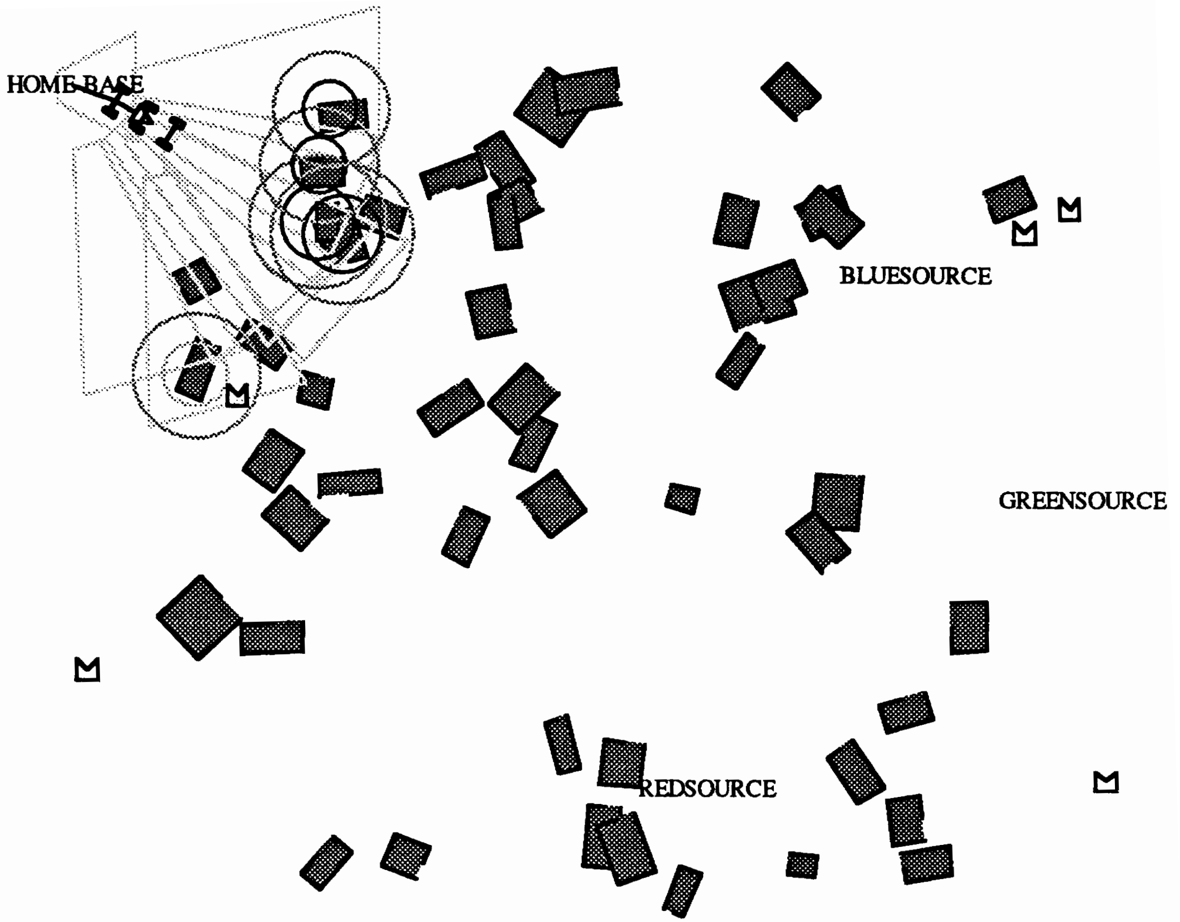


Figure 5.4-5: The robot gets a photograph of a martian.

The robot then proceeds to pick up blue rocks, opportunistically snapping pictures of martians along the way. (See figure 5.4-6.)

Figure 5.4-6: The robot arrives at the source of blue rocks.

Because the robot still has fuel and storage space available, it goes on to collect green rocks rather than return with the blue rocks. However, there is not enough fuel to collect the red sample and still make it back to the home base. The robot therefore returns home to refuel before going on to the red rocks. On the way back it has two collision with boulders that were not detected by the vision system because of noise. Both of these failures are detected and recovered from. (See figure 5.4-7.)

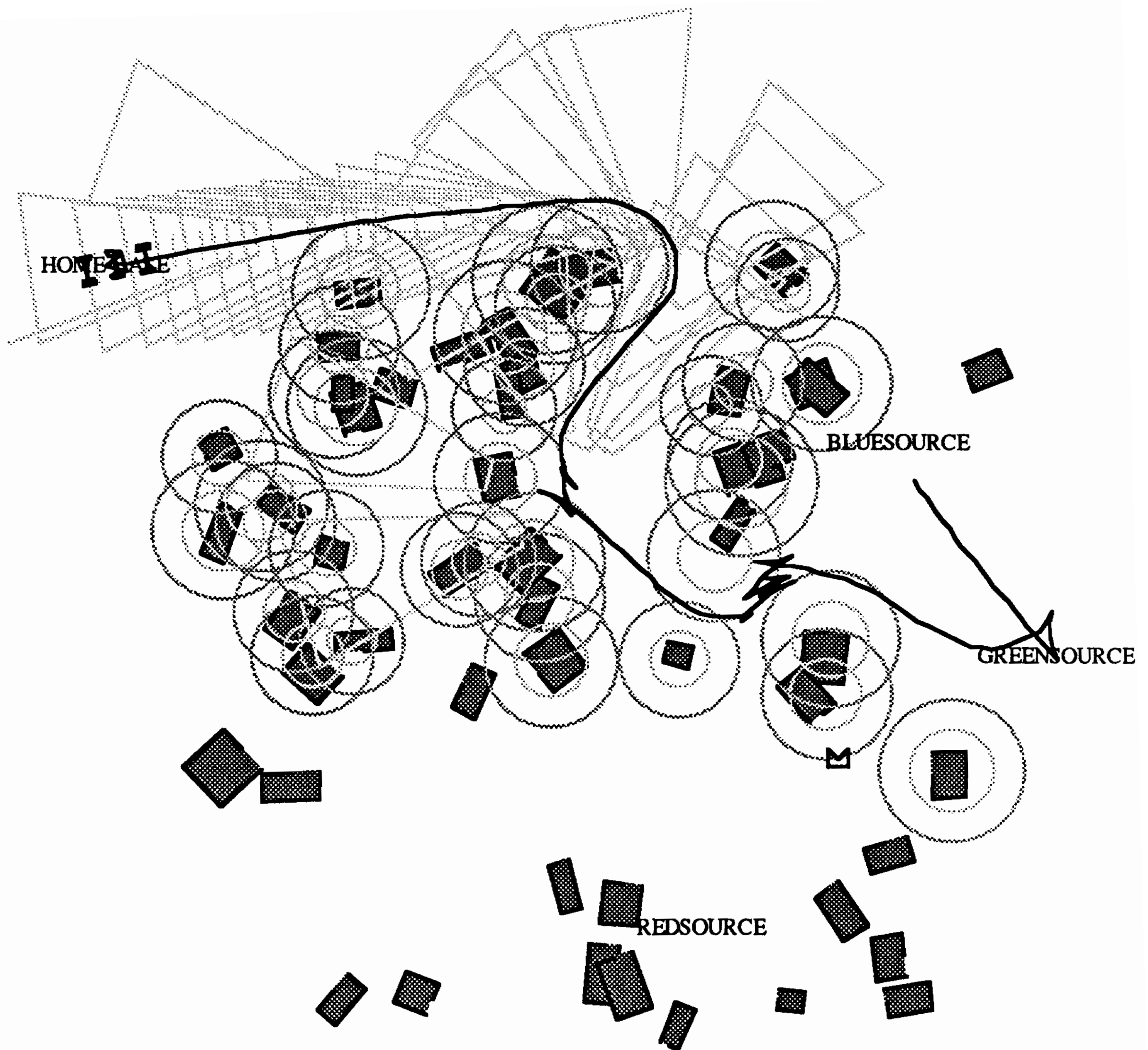


Figure 5.4-7: The robot returns home to deliver rocks and refuel after collecting blue and green rocks.

This robot goes on the collect red rocks. (See figure 5.4-8.) On the way, the robot collides with a boulder which was not detected by the vision system due to noise. This turns out to block the only passage along the robot's original route. As a result, the robot had to drastically change its route. This is an interesting example of a low-level failure causing a major change in the high-level computations.

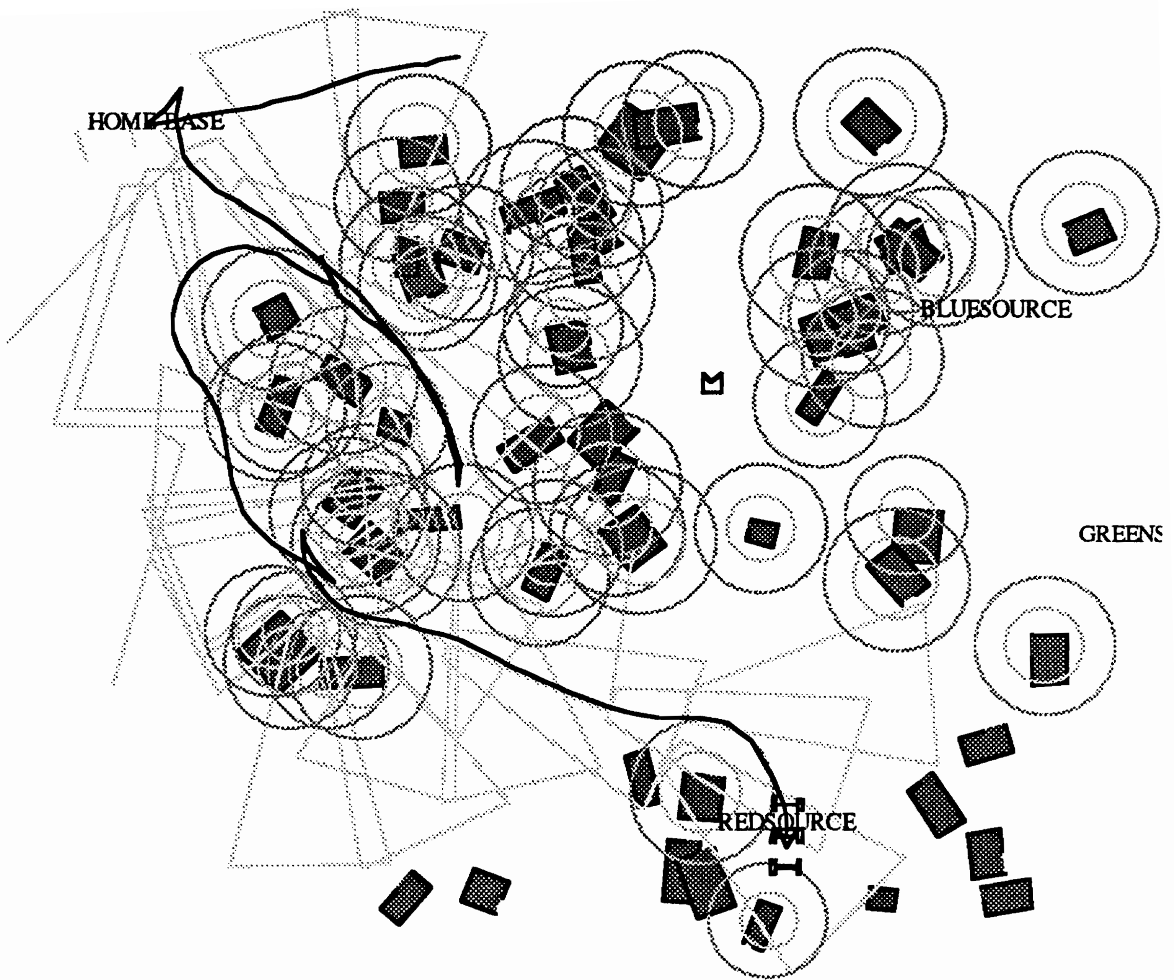


Figure 5.4-8: A low-level failure can cause major changes in the robot's route.

As a dramatic illustration of the difficulty in assessing the performance of ATLANTIS, compare the behavior of the system in this case with the run shown in the appendix. In the latter case, the robot manages to squeeze through the narrow corridor on the way to the red rocks, despite the fact that both runs were started in the same state *with identical random number seeds*. The qualitatively divergent behavior can only be accounted for by the slight variations in timing caused by background processes on the computer. Since the simulator operates in real time, these tiny variations can lead to very different behavior.

5.4.4 Experiment 4: Corridors

Experiment 3 demonstrates robust navigation. However, the algorithm used often sends the robot into unknown areas resulting in inefficient paths due to unexpected obstacles being encountered. To demonstrate the ease with which new deliberative computations can be integrated into the system, a simple topological planner was added which kept track of the locations the robot had visited and tended to make the robot take paths which it had taken before.

Figure 5.4-9 illustrates the system's original performance. The robot's task is to deliver a single blue rock to the home base. It starts by going to where the blue rocks are (adjusting its path for an unseen boulder along the way). When it comes time to make the return trip, it goes the other way around the central obstacle because it assumes that unknown areas are clear. In this particular case the robot makes it home. However, if the other way around the central obstacle had been significantly longer than the first, the robot might have taken a long, unnecessary detour.

One way to solve this problem is for the robot to keep track of where it has been, and to prefer paths that traverse known regions over ones that traverse unknown regions. The robot already does this to a certain extent by keeping track of the areas it has scanned with its camera. However, keeping track of regions as regions can be quite expensive. In fact, the robot can only keep track of about thirty vision frames before its real-time performance begins to suffer.

Another way to accomplish the same thing with less overhead is for the robot to keep track of where it has been in terms of features that are significant to the robot's low-level behaviors [Mataric90], [Kuipers88], [Levitt87], [Thorpe90], [Fennema88]. In this case, the feature that was chosen was narrow passageways between obstacles. A computational process was installed which checked to see when the robot was between two boulders. When it was, the area was marked as a corridor. The points at which the robot entered and exited the corridor were recorded. A simple strategic planner was written which planned paths in terms of transitions between corridors.

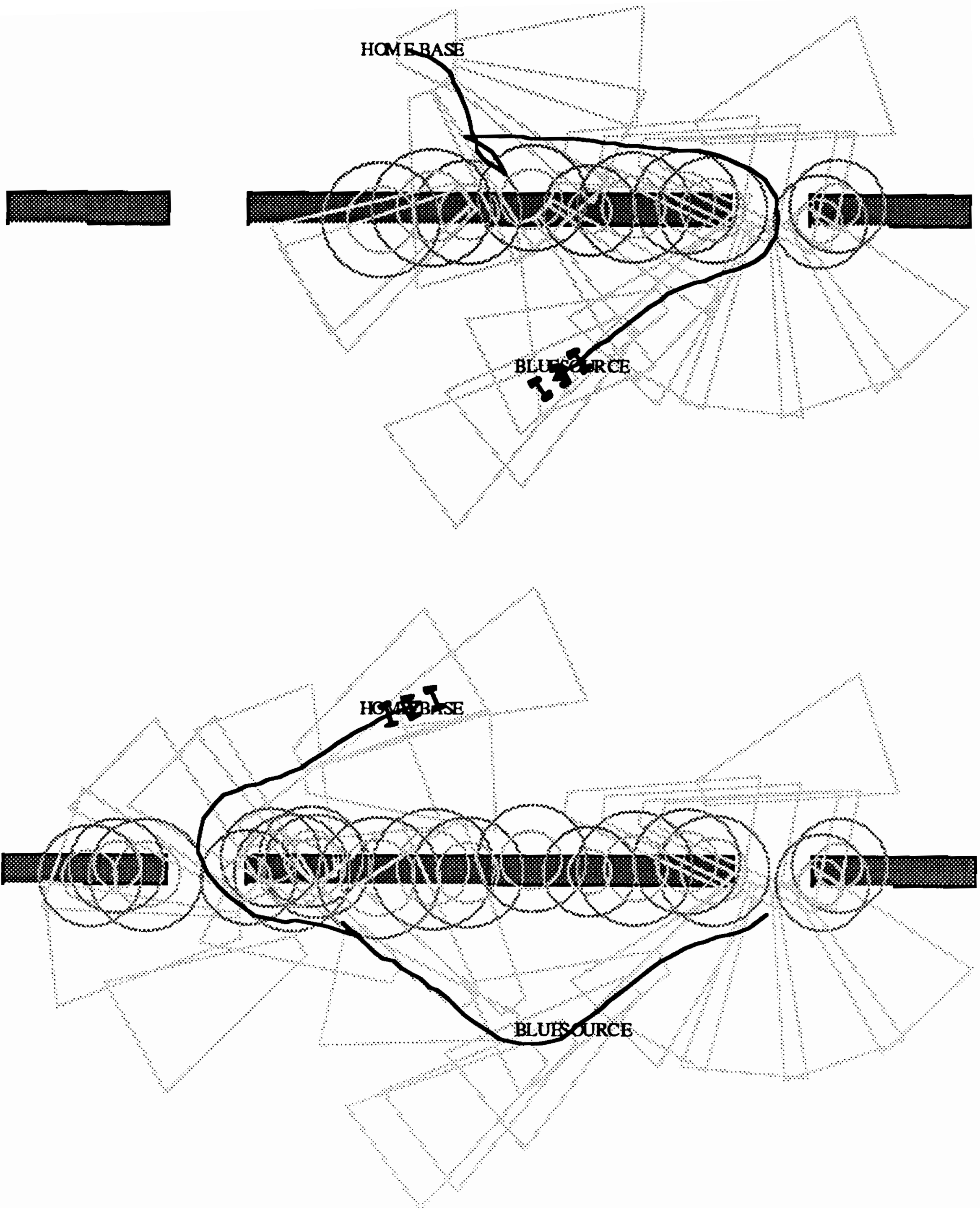


Figure 5.4-9: Treating unknown areas as clear can be risky.

The result was the behavior shown in figure 5.4-10. This run starts the same as the first. However, on the return trip, the robot has a record of the corridor it went through on the way out, and so it prefers this route over the apparently shorter (given its current knowledge of the world) alternate route. The output of the corridor planner is simply a set of constraints on the spins of the NaTs that surround the corridor.

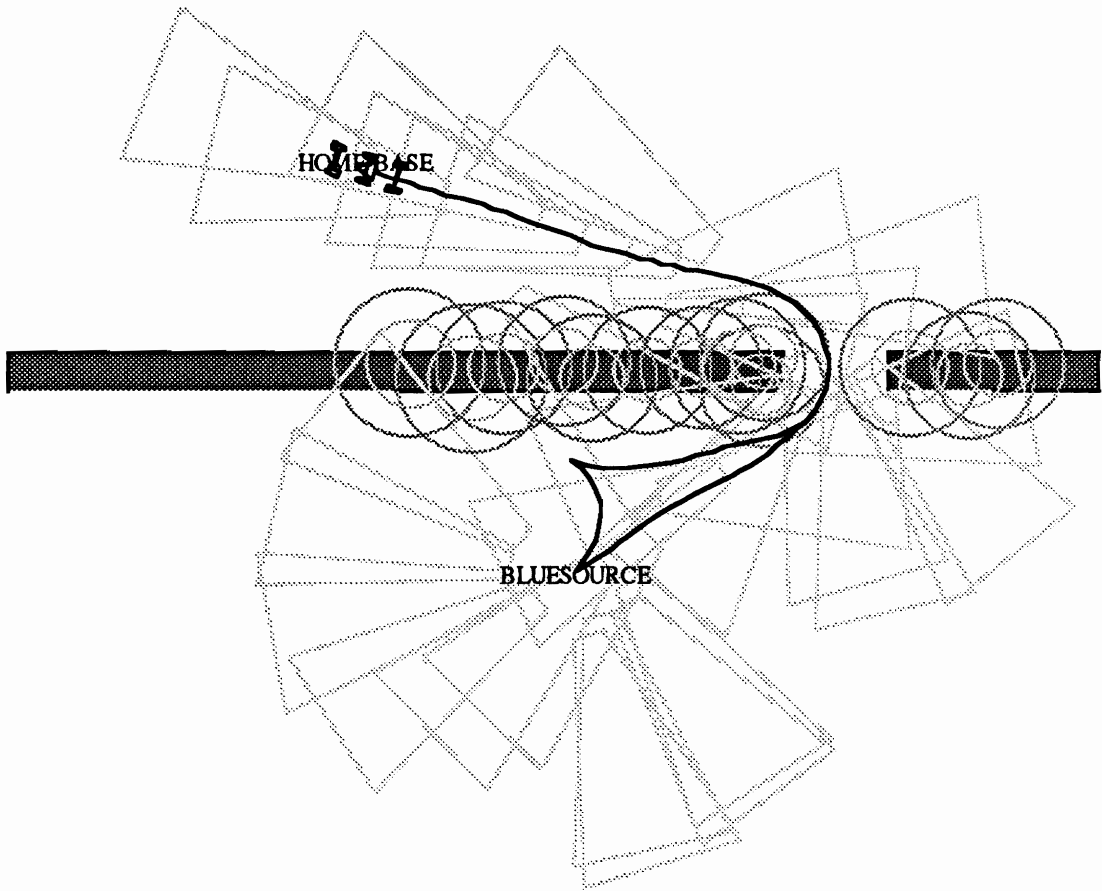


Figure 5.4-10: Remembering corridors leads to more conservative paths.

This experiment demonstrates two key features of the system. First, new behaviors can be easily created by incorporating new deliberative computations using existing technologies. Second, the behavior of the low-level control can provide the input for the deliberative computations, resulting in no additional sensory requirements, and little

additional computation. These deliberative computations can be performed using existing computational technology as long as they are done asynchronously.

5.5 Summary

ALFA and ATLANTIS[6] have been successfully used to control three real robots and a simulated robot performing complex tasks in real-world or simulated real-world environments.

ALFA was used to control a small indoor robot performing an object collecting task. This robot consistently displays robust, goal-directed behavior in the face of noisy sensors, an unpredictable environment, and (extremely) limited computation.

ALFA and a rudimentary sequencer were used to control an indoor robot performing a complex navigation task. The indoor robot was able to navigate from inside a cluttered office to another office on the other side of the building. It was able to do this despite noisy sensors, unexpected obstacles, and poor dead reckoning.

The sequencing and deliberative layers were used to control an outdoor robot performing a navigation task with multiple goals. This system used an existing low-level control system augmented with an interface that conformed to the architecture. This robot was able to successfully coordinate two physical activities (aiming the camera and moving the robot) and three computational activities (computing the path gradient, processing vision data, and global planning). In an experiment with multiple goals, the robot was able to replan on the fly when one of its goals turned out to be blocked by obstacles.

The sequencing and deliberative layers were used to control a simulated robot performing multiple, dynamic tasks in a complex environment. In order to lend credence to the simulated results, the simulator was used to reproduce qualitatively the results of the experiment performed on the real outdoor robot. The code used in these two experiments was absolutely identical. The performance of the simulated robot was qualitatively the same as the real robot. The quantitative differences are due to the fact that the simulated

---

[6]Anything That Limits ATLANTIS' Navigation is a Totally Intractable Situation

vision data is different from the actual vision data. The simulated robot was able to detect obstacles from further away, and thus was able to respond sooner than the real robot did.

The simulator was used to explore the behavior of the system in more depth than is possible using a real robot. The simulated robot was programmed to perform multiple, dynamic tasks in a complex environment. The simulator successfully completed over twenty runs, many lasting for several hours, in a complex cluttered environment. The longest single run performed to date lasted for over eight hours.

Chapter 6: Conclusions

This Chapter begins with a sober summary and progresses in stages to wild speculation. In between it will discuss the major lessons to be learned from the work presented in this thesis, and suggest some directions for future research.

6.1 Summary

6.1.1 The Problem

This thesis addressed the problem of how to control autonomous mobile robots in real-world environments. There are three aspects to this problem which set it apart from other problems in Artificial Intelligence: sensor and actuator noise, unpredictable aspects of the environment, and limited computational resources. These are fundamental problems because they cannot be engineered away to the point where they can be ignored. Therefore, a robust architecture for controlling mobile robots must actively deal with these issues.

The canonical example which was used to illustrate these issues was driving a car to an unfamiliar destination. To accomplish such a task a driver must be able to react quickly to the situation at hand, but must also modulate those reactions according to strategic considerations produced by slow, deliberative thought processes such as reading a map. He must do this using computation limited by the real-time constraints of driving. Obstacles must be avoided before they are collided with. Destinations must be reached before deadlines expire. The driver must be able to deal with failures caused by sensor and actuator noise -- a misread street sign, a faulty gas gauge, inclement weather, mechanical malfunctions. He must be able to perform the task despite the fact that many of the situations encountered along the way cannot be predicted in advance.

Of course, all of these issues have to be translated into a robot's terms. A 68000 microprocessor is no match for the human brain. A robot vision system is badly outclassed by the visual cortex. The amount of knowledge that a designer can incorporate into a robot's control system over the span of a Ph.D. thesis is far less than a human gains over the months and years spent learning to get around in the world. The expectations for our

robots must be correspondingly lower than for humans. A robot with sonars is more like a blindfolded person feeling his way around than a driver in a car. Nevertheless, the fundamental issues are the same: sensor noise, limited computation, and unpredictable contingencies.

6.1.2 Activities

The central hypothesis of this thesis is that a successful mechanism for controlling mobile robots must be based on a hierarchical model of continuous activity. Most previous work in autonomous agents has been based on a discrete action model, largely to support research in planning. While arguably suitable as a framework for planning, discrete actions are not a suitable model of physical processes in general. It is extremely difficult to model simultaneous or overlapping actions using operators. It is impossible to stop an operator in the middle to service an unexpected contingency, unless the operator itself contains this functionality in which case it must be imbued with the intelligence of the entire system.

To avoid these problems, the work in this thesis is based on a model of continuous *activities* rather than discrete Actions. Activities are defined as subsets of the computational and physical processes performed by the robot in a given time. Activities may be initiated and terminated by computational processes which are part of other activities. The output of a computational process which initiates or terminates an activity is called a *decision.*

Activities can be arranged in a hierarchy. Higher level activities contain decision-making computational processes which initiate or terminate lower-level activities. At the bottom of the hierarchy are *primitive activities* which contain no decision-making computations. The only computations in primitive activities are those which directly control physical actions.

High-level activities can be distinguished from low-level ones by four characteristics: commit-time, computation time, reliance on internal state, and abstraction level. High level activities produce decisions which commit the robot to a course of action for a longer period of time than lower-level activities. High-level decisions take longer to make than lower-level ones, and rely more on persistent internal state. Finally, high-level decisions are made at a higher level of abstraction, that is, at a lower level of precision, than lower-level decisions.

These characteristics are closely related. Because high-level activities can initiate a large number of lower-level activities, both directly and indirectly, they typically commit the agent to a course of action for long periods of time. The longer an agent is committed to a course of action the more important it is that that course of action be correct, and the more it pays off to expend computational resources insuring that it is correct. Lengthy computations are inherently reliant on internal state, otherwise they would not be lengthy computations. Internal state is only meaningful if the information represented by that state remains true for the duration of the computation. Therefore, the computations performed by high-level activities must be performed at high levels of abstraction because at low levels of abstraction the world changes too quickly for lengthy computations to produce valid results.

6.1.3 ALFA

The problem of translating the activity hypothesis into a working control mechanism for a robot was approached bottom-up, beginning with the control of primitive activities. Primitive activities are activities which contain no decision-making computations, but are instead controlled by tightly coupling sensors and actuators. A computational mechanism for controlling such activities must be able to compute complex, highly nonlinear transfer functions from sensors to actuators, and must have a fast response time.

In order to support the development of such control structures, a programming language called ALFA was designed and implemented. ALFA programs describe networks of computational modules which are connected to each other and to the outside world by means of channels. ALFA has the following features that make it suitable for controlling primitive activities:

- Dataflow semantics allow the response time to be determined at compile time
- A uniform communications model allows smooth integration with higher level systems
- The abstractions provided by the language allow incremental development and debugging

A portable, retargetable compiler for ALFA was implemented. This compiler produces extremely compact and efficient code which requires very little runtime support.

6.1.4 ATLANTIS

ALFA is not suitable for controlling higher level activities since these require the careful maintenance of large amounts of state information. To control higher level activities, a heterogeneous control architecture called ATLANTIS[1] was developed. ATLANTIS consists of three main components: a control layer which controls primitive activities, a sequencing layer which initiates and terminates activities, and a deliberative layer which performs time-consuming deliberative computations. The sequencing and deliberative layers are implemented almost exclusively using existing technologies.

The control layer is a mostly stateless reactive control mechanism which computes highly non-linear high-dimensional transfer functions from sensor inputs to actuator outputs. This layer is nominally programmed in ALFA, but it does not necessarily have to be.

The sequencing layer is loosely based on Firby's Reactive Action Package system, and is modified to support simultaneous execution of tasks. One interesting problem which arises when tasks are allowed to run in parallel is the issue of how to safely abort a low-priority task which must be interrupted by a high-priority task. A partial solution used by ATLANTIS is an unwind-protect construct which allows an interrupted task to execute a clean-up procedure before relinquishing control of the task queue. A complete solution to this problem is an open research issue.

The deliberative layer is responsible for performing time-consuming deliberative computations in service of high-level activities. It communicates its results to the sequencing layer by means of a shared data base. Computational processes in the deliberative layer are initiated and terminated by the sequencing layer in the same way that primitive activities are initiated and terminated in the control layer.

---

[1]ATLANTIS: a Truly Lively Acronym; Never The Identical Story

6.1.5 Experiments

ALFA was used to control Tooth, a small indoor robot, performing an object-collecting task. This experiment demonstrated that ALFA could be used to control a robot performing a fairly complex task in the real world. It further demonstrated that ALFA could be compiled to run on a uniprocessor with very little runtime support. The control architecture used on Tooth demonstrated some of the design principles of the ATLANTIS architecture. In particular, it demonstrated a layered control structure where higher layers provided guidance to lower layers which operated at lower levels of abstraction. It also demonstrated a rudimentary example of cognizant failure and recovery.

ALFA in conjunction with a rudimentary sequencing layer was used to control Toto, an indoor sonar-based robot, performing a complex navigation task. This experiment demonstrated that ALFA could be easily integrated into systems with traditional deliberative structure.

An implementation of the ATLANTIS sequencing and deliberative layers were used to control Robbie, the JPL planetary rover testbed, performing a complex navigation task. This experiment demonstrated the ability of the architecture to coordinate multiple physical and computational activities, and to use the results of deliberative computations performed at high levels of abstraction to guide the behavior of lower-level activities.

The same code that ran on the planetary rover testbed was used to replicate qualitatively the results of the previous experiment on a real-time simulator. This simulator was then used to perform a variety of complex tasks, all of which involved noisy sensors, limited computation time, and an unpredictable environment. These experiments used existing deliberative technology to provide guidance to lower-level activities at high levels of abstraction. These deliberative computations were, in turn, driven by the demands of the lower-level activities.

6.2 Discussion

The main contribution of this thesis is a methodology and architecture for building robots that work, that is, which display robust goal-directed behavior in real-world environments. Chapter 4 discussed at length the meaning of "robust". To summarize briefly, a robot is robust if a large portion of the time it takes actions which help it achieve

its goals, and seldom takes actions which actively hinder the achievement of its goals. Judging robustness is highly subjective and context-dependent, which is the main reason that only anecdotal evidence can be offered to support the claim. To lend a bit of credence to the anecdotal evidence, the system was implemented on three very different robots, as well as a simulator, and was used to make the robots perform a variety of tasks which have been considered to be quite difficult: sonar-based navigation in an unknown environment, outdoor navigation, and achieving multiple, dynamic goals.

Viewed in this light, the fact that this work contains few proofs is not a serious shortcoming. On the contrary, the absence of proofs is one of its central features! The main result of the thesis is that simple, incomplete, even incorrect algorithms can be combined into an overall system which is more robust than any of its components. This is accomplished by following the design methodology which has been developed in this thesis. The central points of this methodology are:

- Engineer the system bottom-up to engage in activities which fail cognizantly.
- Structure those activities according to the activity hierarchy. Design representations and allocate computing resources accordingly.

This design methodology is a direct analog of commonly accepted software engineering practice. The first point corresponds to the principle that software should be designed bottom-up to detect execution errors. The second point corresponds to the principle that called functions should hide unnecessary details from calling functions.

There are differences, however, between pure software engineering and engineering activities for mobile robots. Computers behave according to mathematical models with an extremely high degree of fidelity. Mobile robots exist in environments which are inherently noisy. The unpredictable aspects of a computer's "environment" are carefully circumscribed by specifying in advance exactly what constitutes valid input. Everything from signal voltages to syntax is defined in advance. Autonomous mobile robots must accept the environment as-is, or with a few minor changes. Putting up a few signs (or UPC codes) to make life easier for our robots is reasonable; expecting every object in the world to be so labelled is not. Finally, computers usually do not operate under the sorts of time pressures that a robot faces. Attempting to calculate the value of π might cause the

computer to miss a deadline, but it won't break the machine. Robots, on the other hand, cannot afford such luxuries. A robot crossing the street must dodge oncoming trucks by hard deadlines or face dire consequences.

It is important, therefore, to recognize that there are two different objectives for the design of a robot control system. The system must make the robot do the right thing whenever possible, but it must also avoid making the robot do the wrong thing. The latter of these is far more critical. Consider a robot in the middle of a street faced with an oncoming truck. If it takes the robot very long to decide which side of the street it should run to, it is better off just choosing a side at random. Even if it later turns out that it chose the wrong side of the street (because, say, its goal is on the other side) the robot is still better off than if it had tried to plan an optimal course of action and, as a consequence, gotten run over by the truck.[2]

Primitive activities, in general, should be responsible for avoiding catastrophic mistakes such as stopping in the way of oncoming trucks, but not for choosing an optimal course of action. The reason for this is that choosing an optimal course of action is almost always more difficult than avoiding catastrophes. This, in turn, is due to the fact that an agent's goals may change, but the sorts of dangers it is likely to encounter tend to stay the same. (Trucks need to be avoided regardless of one's destination.) Thus, the procedures needed to avoid danger can be hardcoded into the control layer.

Once primitive activities have been developed and debugged, they can be used as building blocks for developing higher-level activities. Primitive activities can be treated abstractly in much the same way as operators, except that the interface is different. Where operators are simply invoked, primitive activities need to be separately initiated, monitored, and terminated. Where operators are usually thought of as producing some predictable change in the world, primitive activities are not necessarily predictable. However, whatever changes they do bring about in the world will be reported to the sequencing layer (assuming the activity has been properly designed to fail cognizantly).

Primitive activities can also serve to integrate large amounts of sensory information about the world without the need for explicit deliberative computations. A properly designed primitive activity performs implicit filtering on large amount of input data and can

---

[2]See the preface of [Hammond89] for an entertaining parable on this topic.

provide reliable sensory information to the sequencing layer at a high level of abstraction. An example using a wall-following activity to construct a topological map of the environment was demonstrated on a real robot.

In addition to the problems addressed by Firby in his work on RAPs, the sequencing layer must address two additional issues which arise from the activity model of actions. First, a system of semaphores must be provided to prevent primitive activities with conflicting resource demands from overlapping. Second, some sort of mechanism must be provided to allow a clean-up procedure to be executed by a low-priority activity when it is interrupted by a high-priority activity (the Wesson Oil problem).

The quality of the system's decisions can often be greatly improved by performing some time-consuming deliberative computations explicitly. From the system's point of view, such computations can be viewed as time-consuming sensory activities. Thus, they should occur asynchronously, and be directed by the sequencing layer which initiates and terminates deliberative computations in the same that it initiates and terminates primitive activities. Deliberative computations should be performed at a high level of abstraction, and the results should be used to guide but not to control the robot's actions.

In a way, this thesis is a vindication of both sides of the ongoing planning vs. reactivity controversy. On the one hand, simple reactive behaviors such as those advocated by Brooks and Connell are necessary in order to respond to situations in real time. On the other hand, planning and other deliberative computations are necessary in order to guide the system's actions towards goals. This thesis has attempted to show that these approaches are not at odds as may have been previously supposed. As one attempts to push the reactive approach towards higher levels of functionality one ends up with a planner. As one attempts to push planning down to the level of detail at which real robots must operate one ends up with a reactive controller. A sequencer serves as the glue to bind these two approaches together into a working system.

6.3 Future Work

This thesis leaves open a large number of interesting research issues. Some of these are discussed below.

Extensions

The most immediate need for more research is to test the ATLANTIS architecture and design methodology on a larger variety of robots performing a larger variety of tasks. To do this, the structure of the architecture should be more clearly defined. The ATLANTIS sequencing layer currently does not enforce restrictions on the computations that it performs. Task schema methods are currently arbitrary lambda expressions. A syntax for programming the sequencing layer should be defined. The RAP description language would be a good starting point, but it relies heavily on the discrete-action model. Ideally, this syntax should get compiled down to machine code in the same way that ALFA is.

Manipulation

In order for mobile robots to be truly useful they need to be able to manipulate objects in the world as well as getting from place to place. This thesis has left the manipulation problem completely untouched. It would be interesting to see if the ATLANTIS approach extends to manipulation. Some work in this direction is already being done (e.g., [Bao90], [Kheradpir88]).

Learning

ATLANTIS provides interesting avenues for research in machine learning. In general, learning in ATLANTIS would consist of transferring functionality from one layer to another. The deliberative layer, for example, might be able to encapsulate the results of its computations as task schemas which could then run in the sequencing layer. A task schema whose response time was too slow might get compiled down to an ALFA module and run in the control layer.

Learning might also be used to propagate information in the other direction. For example, suppose that one of the modules in the control layer had a bug in it. The system could modify the task schemas in the sequencing layer so that they no longer used this module. Perhaps the bug could even be diagnosed and repaired. The fact that ATLANTIS is a situated system, that is, that it exists in a real environment, provides innumerable opportunities for the system to diagnose and modify its own behavior.

Managing Resources

A better mechanism for handling resource allocation and dealing with resource conflicts needs to be devised. The simple semaphore-based scheme used in this thesis is not adequate for many applications [Miller85]. Avoiding problems arising from resource constraints in an embedded system is a virtually untouched area of research.

A better solution to the Wesson Oil problem needs to be found. To review briefly, this is the problem of what to do when a high-priority activity needs to interrupt a low-priority activity because of resource conflicts. In many cases the low-priority activity needs to be able to execute a clean-up procedure before it can be aborted safely. In the case of ATLANTIS, this usually means terminating any primitives that have been initiated by the low-priority activity. This is accomplished in ATLANTIS by installing an unwind-protect mechanism into the sequencing layer which simply calls a fixed clean-up procedure associated with an activity whenever that activity must be aborted. However, this is only a partial solution to the problem. In many cases, the proper clean-up procedure is context-dependent, as it is in the case of the chicken-frying example that gives the problem its name. The stove needs to be turned off if the child must be taken to the hospital, but if all the kid needs is a band-aid the stove might be safely left burning. Sometimes this decision needs to be made very quickly, e.g. if a burglar enters the kitchen.

One possible approach to this problem is to use a probabilistic query-directed projection mechanism like the one described in [Hanks90b]. A query-directed projection mechanism predicts only as much of the future as is necessary to answer the specific question posed to it. If the questions are sufficiently vague, the response time can be quite fast. A query-directed projection mechanism could conceivably determine very quickly that there is more to be gained by running away from a burglar than by turning off the stove.

Planning

Another open research area is the development of incremental transformational planners. A transformational planner [Hanks90a], [McDermott90] incrementally improves a plan as it runs. Such a planner would be an example of an anytime algorithm: at every stage in the algorithm it has a workable plan that can be produced upon demand [Boddy89]. The longer the algorithm runs, the better the plan gets (presumably). Any deliberative algorithm run asynchronously can be considered a degenerate case of an

anytime algorithm whose utility starts at 0 (no answer) and at some later point in time increases to some positive value when the computation finishes. Since the system is structured in this way, it would be extremely useful to design algorithms that could take better advantage of this structure. (The task planner in the Martian experiment was a toy incremental planner.)

Better Simulators

Simulators have been much maligned because they have been much abused. Nevertheless, a good simulator can be an invaluable tool for developing and debugging robot control algorithms. There are a number of challenging problems to be addressed here. The ideal simulator would allow the user to construct customized robots by mixing and matching "sensors" and "actuators" which are actually software objects. The simulator should allow new sensor and actuator modules to be defined in a way that makes it possible to simply "install" the resulting objects into a simulated robot. There are many challenging software engineering and language design issues here. For example, how does one model the interaction between the physical extent of the robot and objects in the world? How does one model the interaction of "actuators" with objects in the world and with each other? How could one model the interference between active sensors on multiple robots? What if one robot tried to grasp another with a simulated gripper? Comparing this wish list with just about any modern simulator (e.g., [LePape90]) will reveal that there is much yet to be done.

No simulator can model all of the interesting aspects of the world. Reality is simply much too complex for that. The interesting problem is how to model enough of the world to make the behavior of control programs reflect their behavior on real robots while keeping the simulation fast enough to be useful.

6.4 Final Words

AI has historically been a field with a large gap between theory and practice. Part of the problem is that there is no concensus about what even constitutes the field. In the early days, solving differential equations was considered AI. Because solving differential

equations is typically harder for people than, say, driving to Hoboken, it was thought that, once computers could solve differential equations, driving to Hoboken would be easy.

Unfortunately, it hasn't turned out that way. Computers can now solve differential equations better than people can, but they still can't drive to Hoboken (despite the work presented here!). Part of the problem is that the symbolic processing model on which most of AI is based turns out to be a very good model of solving differential equations, but not a good model of driving to Hoboken.

Unfortunately, driving to Hoboken is a physical phenomenon of astounding depth and complexity, and thus it is exceedingly difficult to formalize. It may not even be possible to do so. Some phenomena of nature, it is now becoming clear, cannot be characterized mathematically except as statistical approximations. Quantum mechanics and turbulence are two such phenomena. (The opening quotation of this thesis describes another.)

Whether driving to Hoboken belongs in this category remains to be seen. Certainly the dearth of hard results (at least by comparison with differential-equation solving) after twenty years of theorizing suggests that it might be. However, in order to determine this, the phenomenon must first be isolated in a reproducible way so that it can be studied. One way to do this is to observe humans driving to Hoboken and performing other real-world tasks. This is the approach taken by Suchman [Suchman87] and Agre [Agre88] among others. Another way is to attempt to reproduce the phenomenon on a machine in the absence of a complete mathematical understanding of it.

ATLANTIS[3] reproducibly produces interesting and complex behavior in mechanical systems operating in unstructured environments. If nothing else, the value of the work is that it provides a concrete example of an interesting phenomenon for people to argue about.

---

[3]All Too Late, A Nifty Title Is Sought

Bibliography

[Agre88] Phillip E. Agre, "The Dynamic Structure of Everyday Life," Technical Report 1085, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1988.

[Agre90] Phillip E. Agre and David Chapman, "What are Plans for?" *Robotics and Autonomous Systems*, vol. 6, pp. 17-34, 1990.

[Allen83] James Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, vol. 26, pp. 832-843, 1983.

[Anderson90] Tracy Anderson and Max Donath, "Animal Behavior as a Paradigm for Developing Robot Autonomy," Robotics and Intelligent Systems, vol. 6, pp. 145-168, 1990.

[Andress88] K. M. Andress and A. C. Kak, "Evidence Accumulation and Flow of Control in a Hierarchical Spatial Reasoning System," *AI Magazine*, vol. 9, no. 2, Summer 1988.

[Angle89] Colin Angle, "Tooth Docs: the Paper", unpublished manuscript.

[Arkin90] Ronald C. Arkin, "Integrating Behavioral, Perceptual and World Knowledge in Reactive Navigation," *Robotics and Autonomous Systems*, vol. 6, pp. 105-122, 1990.

[Bao90] C. Bao and H. van Brussel, "A Sensory Controlled Gripper System," *Robotics and Autonomous Systems*, vol. 6, pp. 283-295, 1990.

[Barraquand89] Jérôme Barraquand and Jean-Claude Latombe, "On Nonholonomic Mobile Robots and Optimal Maneuvering," *Proceedings of the IEEE International Symposium on Intelligent Control*, 1989.

[Biber80] C. Biber, *et al.*, "The Polaroid Ultrasonic Ranging System," *Proceedings of the 67th Conference of the Audio Engineering Society*, 1980.

[Boddy89] Mark Boddy and Thomas Dean, "Solving Time-Dependent Planning Problems," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1989.

[Boyce77] William E. Boyce and Richard C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, New York: John Wiley and Sons, 1977.

[Braitenberg84] Valentino Braitenberg, *Vehicles: Experiments in Synthetic Psychology*, Cambridge, Massachusetts: MIT press, 1984.

[Braunegg90] David J. Braunegg, "MARVEL: A System for Recognizing World Locations with Stereo Vision," Technical Report 1229, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1990.

[Brooks86] Rodney A. Brooks, "A Robust Layered Control System for a Mobile Robot", *IEEE Journal on Robotics and Automation*, vol. RA-2, no. 1, March 1986.

[Brooks89] Rodney Brooks, "A Robot that Walks: Emergent Behaviors from a Carefully Evolved Network," *Neural Computation*, vol. 1, pp. 253-262, 1989.

[Brooks90] Rodney Brooks, "The Behavior Language User's Guide," MIT AI Lab memo 1127, 1990.

[Broverman87] Carol Broverman and W. Bruce Croft, "Reasoning About Exceptions During Plan Execution", *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1987.

[Chapman87] David Chapman, "Planning for Conjunctive Goals", *Artificial Intelligence*, vol. 32, no. 3, pp. 333-378, July 1987.

[Chapman89] David Chapman, "Penguins can make Cake", *AI Magazine*, vol. 10, no. 4, pp. 45-50, Winter 1989.

[Connell89] Jonathan Connell, "A Colony Architecture for an Artificial Creature", Technical Report 1151, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1989.

[Connell90] Jonathan Connell and Paul Viola, "Cooperative Control of a Semi-Autonomous Mobile Robot," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1990.

[Dean88] Thomas Dean, R. James Firby and David P. Miller, "Hierarchical Planning with Deadlines and Resources," *International Journal of Computational Intelligence*, vol. 4, pp. 381-398, 1988.

[Dean90a] Thomas Dean, et al., "Coping with Uncertainty in a Control System for Navigation and Exploration," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1990.

[Dean90b] Thomas Dean, Kenneth Bayse and Moises Lejter, "Planning and Active Perception," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Durfee90] Edmund H. Durfee, "A Cooperative Approach to Planning for Real-Time Control," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Dybvig89] R. K. Dybvig and R. Hieb, "Engines from Continuations," *Journal of Computer Languages*, vol. 2, pp. 109-124, 1989.

[Erdmann90] Michael Erdmann, "On Probabilistic Strategies for Robot Tasks," Technical Report 1155, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1990.

[Everett89] H. R. Everett, "Survey of Collision Avoidance and Ranging Sensors for Mobile Robots," *Robotics and Autonomous Systems*, vol. 5, pp. 5-67, 1989.

[Fennema88] Claude L. Fennema Jr., Edward M. Riseman, and Allen R. Hanson, "Planning with Perceptual Milestones to Control Uncertainty in Robot Navigation," SPIE Sensor Fusion, 1988.

[Firby89] R. James Firby, "Adaptive Execution in Complex Dynamic Worlds," Technical Report YALEU/CSD/RR#672, Yale University, 1989.

[Gat90] Erann Gat, et al., "Path Planning and Execution Monitoring for a Planetary Rover," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1990.

[Georgeff87] Michael Georgeff and Amy Lanskey, "Reactive Reasoning and Planning," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1987.

[Haber90] Ralph Norman Haber, "Why Mobile Robots Need a Spatial Memory," *Proceedings of the SPIE Conference on Sensor Fusion III*, vol. 1383, 1990.

[Hammond89] Kristian Hammond, *Case-Based Planning*, New York: Academic Press, 1989.

[Hammond90] Kristian Hammond, Timothy Converse and Mitchell Marks, "Toward a Theory of Agency," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Hanks90a] Steve Hanks and R. James Firby, "Issues and Architectures for Planning and Execution," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Hanks90b] Steve Hanks, "Practical Temporal Projection," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1990.

[Hendler90] James Hendler, Austin Tate, and Mark Drummond, "AI Planning: Systems and Techniques," AI Magazine, vol. 11, no. 2, Summer 1990.

[Hogge88] John Hogge, "Prevention Techniques for a Temporal Planner," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.

[Horswill88] Ian Horswill and Rodney Brooks, "Situated Vision in a Dynamic World: Chasing Objects," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.

[Ingrand90] François Félix Ingrand and Michael P. Georgeff, "Managing Deliberating and Reasoning in Real-Time AI Systems," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Kaelbling88] Leslie Pack Kaelbling, "Goals as Parallel Program Specifications", *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.

[Kaelbling90] Leslie Kaelbling and Stanley Rosenschein, "Action and Planning in Embedded Agents," *Robotics and Autonomous Systems*, vol. 6, pp. 35-48, 1990.

[Kheradpir88] Shaygan Kheradpir and James S. Thorp, "Real-Time Control of Robot Manipulators in the Presence of Obstacles," *IEEE Journal of Robotics and Automation*, vol. 4, no. 6, 1988.

[Kuipers88] B. Kuipers and Y. T. Byun, "A Robust Qualitative Method for Spatial Learning in Unknown Environments," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.

[Laird91] John Laird, *et al.*, "Robo-Soar: An Integration of External Interaction, Planning and Learning using Soar," *Robotics and Autonomous Systems*, in press.

[LePape90] Claude LePape, "Simulating Actions of Autonomous Agents: an Overview," *Proceedings of the AAAI Workshop on Artificial Intelligence and Simulation*, 1990.

[Levitt87] Tod S. Levitt, *et al.*, "Qualitative Landmark-Based Path Planning and Following," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1987.

[Maes90] Pattie Maes and Rodney Brooks, "Learning to Coordinate Behaviors," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1990.

[Malkin90] Peter K. Malkin and Sanjaya Addanki, "LOGNets: A Hybrid Spatial Representation for Robot Navigation," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1990.

[Mataric90] Maja Mataric, "A Distributed Model for Mobile Robot Environment Learning and Navigation," Technical Report 1228, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1990.

[Matthies91] Larry Matthies, "Stereo Vision for Planetary Rovers: Stochastic Modeling to Near-Real-Time Implementation, JPL Technical Report D-8131, 1991.

[McDermott90] Drew McDermott, "Planning Reactive Behavior: A Progress Report," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Miller84] David P. Miller, "Two-Dimensional Mobile Robot Positioning Using Onboard Sonar," *Proceedings of the IEEE Pecora 9 Conference on Spatial Information Technologies for Remote Sensing Today and Tomorrow*, 1984.

[Miller85] David P. Miller, "Planning by Search Through Simulations", Technical Report YALEU/CSD/RR#423, Yale University. 1985.

[Miller86] David P. Miller, "Scheduling Robot Sensors for Multiple Sensory Tasks," *Proceedings of Ultratech-Robots West*, 1986.

[Miller89] David P. Miller, "Execution Monitoring for a Mobile Robot System," *Proceedings of the SPIE Conference on Intelligent Control and Adaptive Systems*, vol. 1196, pp. 36-43, Philadelphia, PA, November 1989.

[Miller91] David P. Miller and Marc G. Slack, "Global Symbolic Maps from Local Navigation," *Proceedings of the 1991 National Conference on Artificial Intelligence (AAAI)*, Anaheim, CA, July 1991.

[Mitchell90] Tom Mitchell, "Becoming Increasingly Reactive," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1990.

[Nilsson80] Nils J. Nilsson, *Principles of Artificial Intelligence*, Palo Alto, California: Tioga Publishers, 1980.

[Noreils90] Fabrice Noreils, "Integrating Error Recovery in a Mobile Robot Control System," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1990.

[Ogata87] Katsuhiko Ogata, *Discrete-Time Control Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[Payton86] David W. Payton, "An Architecture for Reflexive Autonomous Vehicle Control," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986.

[Payton90a] David Payton, "Exploiting Plans as Resources for Action," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Payton90b] David Payton, J. Kenneth Rosenblatt and David Keirsey, "Plan-Guided Reaction," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, pp. 1370-1382, 1990.

[Rosenblatt89] J. Kenneth Rosenblatt and David Payton, "A Fine-grained Alternative to the Subsumption Architecture for Mobile Robot Control," *Proceedings of International Joint Conference on Neural Networks*, Washington D.C., June, 1989.

[Rosenschein86] Stanley Rosenschein and Leslie Kaelbling, "The Synthesis of Digital Machines with Provable Epistemic Properties," Technical note 412, SRI Artificial Intelligence Center, 1986.

[Sarachik89] Karen Sarachik, "Characterizing an Indoor Environment with a Mobile Robot and Uncalibrated Stereo," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1989.

[Schoppers87] M. J. Schoppers, "Universal Plans for Reactive Robots in Unpredictable Domains," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1987.

[Shiller90] Zvi Shiller and Hsueh-Hen Lu, "Optimal Motion Planning of Autonomous Vehicles in Three-Dimensional Terrains," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1990.

[Simmons90] Reid Simmons, "An Architecture for Coordinating Planning, Sensing and Action," *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Slack90] Marc G. Slack, "Situationally Driven Local Navigation for Mobile Robots", JPL Publication 90-17, California Institute of Technology Jet Propulsion Laboratory, April 1990.

[Slade87] Stephen Slade, *The T Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[Smith89] D. B. Smith and J. R. Matijevic, "A System Architecture for a Planetary Rover," *Proceedings of the NASA Conference on Space Telerobotics*, vol. 1, JPL Publication 89-7 California Institute of Technology Jet Propulsion Laboratory, 1989.

[Soldo90] Monnett Soldo, "Reactive and Preplanned Control in a Mobile Robot," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1990.

[Stentz90] Anthony Stentz, "The Navlab System for Mobile Robot Navigation", Ph.D. Dissertation, Carnegie Mellon University School of Computer Science, 1990.

[Suchman87] Lucy Suchman, *Plans and Situated Actions*, Cambridge, England: Cambridge University Press, 1987.

[Thorpe90] Charles Thorpe and Jay Gowdy, "Annotated Maps for Autonomous Land Vehicles," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 1990.

[Ullman84] Shimon Ullman, "Visual Routines," *Cognition*, vol. 18, pp. 97-159, 1984.

[Wehner87] Rüdiger Wehner, "Matched Filters: Neural Models of the External World," *Journal of Comparative Psychology*, vol. 161, pp. 511-531, 1987.

Appendix: Annotated Run

This section presents an edited, annotated transcript of a complete run of the ATLANTIS system performing rock-delivery and martian-chasing tasks. The raw transcript runs well over 100 pages, so many of the uninteresting sections have been removed. Such deletions are noted when they occur. Also, dumps of the world state have been removed and replaced with figures.

The initial configuration is shown in figure A-1. The robot's tasks are to collect five blue rocks, three red rocks, and two green rocks and return them to the home base. The robot also must photograph each of the five martians, shown as M-shaped figures.

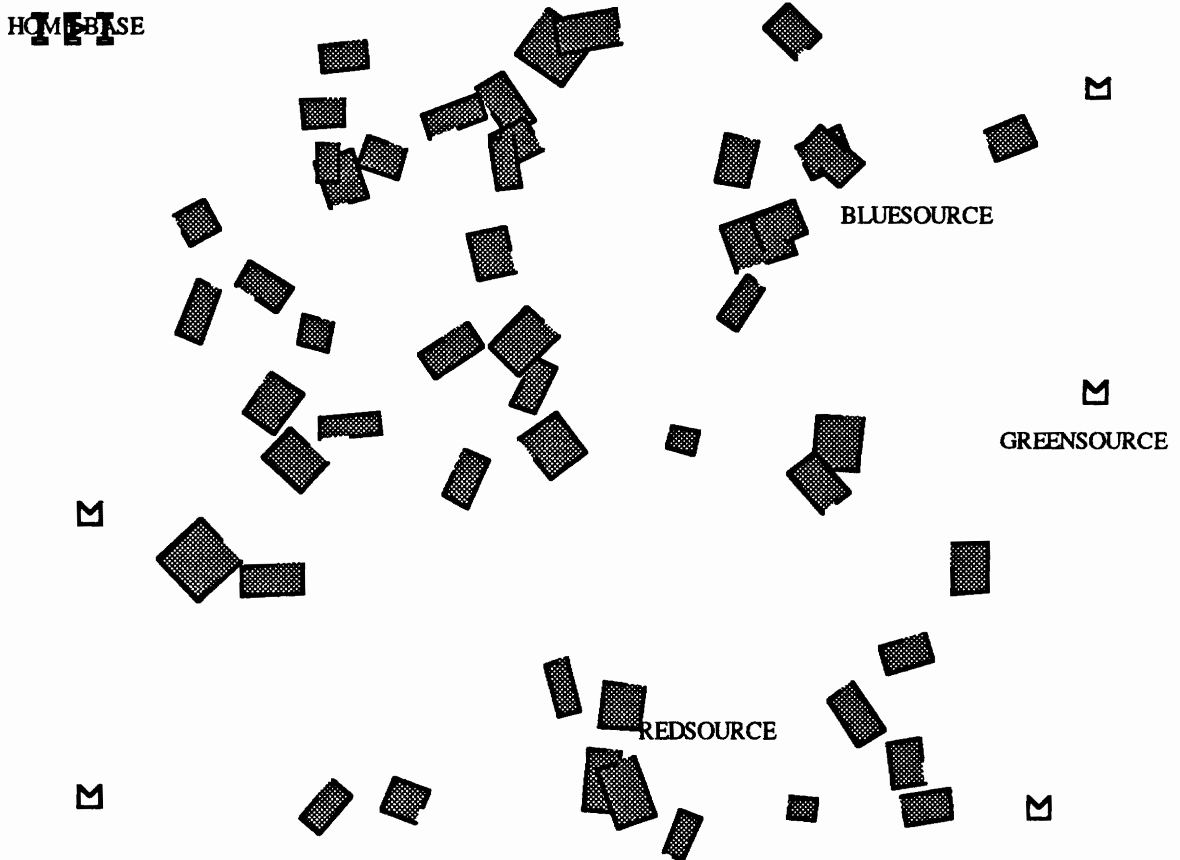


Figure A-1: The initial situation.

At the start of the run the robot knows nothing about the world except the location of the rock sources and the martians. It begins by formulating an initial plan, since it can do

nothing until it has decided where it should go first. It also initiates a vision scan of the area immediately in front of it, since it cannot move until this area has been scanned. All of the text in monaco typeface was produced by the ATLANTIS sequencer.

```
? (robot-go robbie)
Data base contains the following:
#<Assertion (ROBOT-TASKS (BLUE 5 HOME-BASE) (RED 3 HOME-BASE)
(GREEN 2 HOME-BASE))>

Beginning cycle 1.  Robot status is :STOPPED.
Time is 64520.68
Robot is at x=-25.000 y=20.000 h=0.000.
Telemetry: :BOULDER-DATA
Current plan: LISP:NIL
*** Waiting for a plan. ***
*** Waiting for vision data ***
Aiming camera.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 2.  Robot status is :STOPPED.
Time is 64524.23
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.66531364752196 -
3.161318146305735) #<A Martian named FRED>)
*** Setting goal (-11.66531364752196 -3.161318146305735).
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.0.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.
```

The robot has decided to chase Fred, the martian, because there is a clear path to rendezvous with Fred as far as the robot can tell given its current knowledge of the world. The robot sets its current goal to the rendezvous point. However, it has to wait until enough of the surrounding area has been scanned before it can begin to move.

```
Beginning cycle 3.  Robot status is :STOPPED.
Time is 64530.58
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.66531364752196 -
3.161318146305735) #<A Martian named FRED>)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 4.  Robot status is :STOPPED.
Time is 64532.77
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.66531364752196 -
3.161318146305735) #<A Martian named FRED>)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.
```

Note that the robot uses the time while waiting for the camera to move into position to update its global plan. It continues with its original intent to chase Fred, but the location of the rendezvous point is updated.

```
Beginning cycle 5.  Robot status is :STOPPED.
Time is 64535.08
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.20192083970617 -2.64783494066797)
#<A Martian named FRED>)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 6.  Robot status is :STOPPED.
Time is 64538.25
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.20192083970617 -2.64783494066797)
#<A Martian named FRED>)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 7.  Robot status is :STOPPED.
Time is 64540.43
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-11.20192083970617 -2.64783494066797)
#<A Martian named FRED>)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 8.  Robot status is :STOPPED.
Time is 64542.92
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.8992001889894 -2.312391732398357)
#<A Martian named FRED>)
*** Setting goal (-10.8992001889894 -2.312391732398357).
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 9.  Robot status is :STOPPED.
Time is 64550.30
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.8992001889894 -2.312391732398357)
#<A Martian named FRED>)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 10.  Robot status is :STOPPED.
Time is 64552.65
```

```
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.8992001889894 -2.312391732398357)
#<A Martian named FRED>)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 11.  Robot status is :STOPPED.
Time is 64555.07
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.41884734676741 -
1.780115199182518) #<A Martian named FRED>)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 12.  Robot status is :STOPPED.
Time is 64558.30
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.41884734676741 -
1.780115199182518) #<A Martian named FRED>)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 13.  Robot status is :STOPPED.
Time is 64560.75
Robot is at x=-25.000 y=20.000 h=0.000.
Current plan: (CHASE-MARTIAN (-10.41884734676741 -
1.780115199182518) #<A Martian named FRED>)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 14.  Robot status is :MOVING.
Time is 64566.18
Robot is at x=-24.861 y=19.980 h=6.130.
Current plan: (CHASE-MARTIAN (-9.44858267315991 12.483300963139)
#<A Martian named MARTIANETTE>)
*** Setting goal (-9.44858267315991 12.483300963139).
Moving robot: ((2.0 0.4026753915781329 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

Note that the robot's global plan has changed. Because the world changed in the time it took to scan the area to the robot could move, it has abandoned chasing Fred and is now trying to rendezvous with Martianette.

```
Beginning cycle 15.  Robot status is :MOVING.
Time is 64577.52
Robot is at x=-24.470 y=19.901 h=6.042.
```

```
Current plan: (CHASE-MARTIAN (-9.229106688780153 12.69491318996489)
#<A Martian named MARTIANETTE>)
Moving robot: ((2.0 0.3636180356866889 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.4026753915781329.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 16.  Robot status is :MOVING.
Time is 64585.58
Robot is at x=-23.859 y=19.711 h=5.924.
Current plan: (CHASE-MARTIAN (-9.229106688780153 12.69491318996489)
#<A Martian named MARTIANETTE>)
Moving robot: ((2.0 0.2477035489029236 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 17.  Robot status is :MOVING.
Time is 64591.13
Robot is at x=-23.452 y=19.543 h=5.867.
Current plan: (CHASE-MARTIAN (-9.229106688780153 12.69491318996489)
#<A Martian named MARTIANETTE>)
Moving robot: ((2.0 0.1918542631157676 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

At this point, the robot has done a number of vision scans and knows much more about the world than it did initially. In particular, it knows that the direct routes to all the martian rendezvous points are blocked by boulders. Thus, the robot gives up on Martianette, and decides to go pick up blue rocks instead.

```
Beginning cycle 18.  Robot status is :MOVING.
Time is 64600.02
Robot is at x=-22.750 y=19.203 h=5.794.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Setting goal (15.0 11.0).
Moving robot: ((2.0 0.096368610674336 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.1918542631157676.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 19.  Robot status is :MOVING.
Time is 64622.13
Robot is at x=-22.311 y=18.965 h=5.780.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 0.1112965106464294 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 20.  Robot status is :MOVING.
Time is 64629.30
```

```
Robot is at x=-21.877 y=18.716 h=5.753.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.
```

Despite the fact that the robot is no longer actively chasing her, Martianette happens to wander into range of the robot's camera, and so the robot takes advantage of the opportunity by re-aiming its camera.

```
Beginning cycle 21.  Robot status is :STOPPED.
Time is 64633.70
Robot is at x=-21.620 y=18.562 h=5.732.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.7388481836422915.
*** Got a photograph of #<A Martian named MARTIANETTE>! ***
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.
```

The world now looks like figure A-2. Note that the vision frame which captured Martianette is slightly off to one side. If the martian had not been there, the robot would have been scanning straight ahead.

After this, nothing much interesting happens for quite a while as the robot moves uneventfully across the upper portion of the figure. Most of this part of the transcript has been deleted, but it is all essentially identical to the last part of the traverse which we pick up at the 87th cycle.

[66 cycles deleted]

```
Beginning cycle 87.  Robot status is :MOVING.
Time is 65109.67
Robot is at x=4.907 y=20.744 h=5.421.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 0.3057310641970341 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 88.  Robot status is :MOVING.
Time is 65117.22
Robot is at x=5.223 y=20.330 h=5.344.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 0.1521904321559679 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

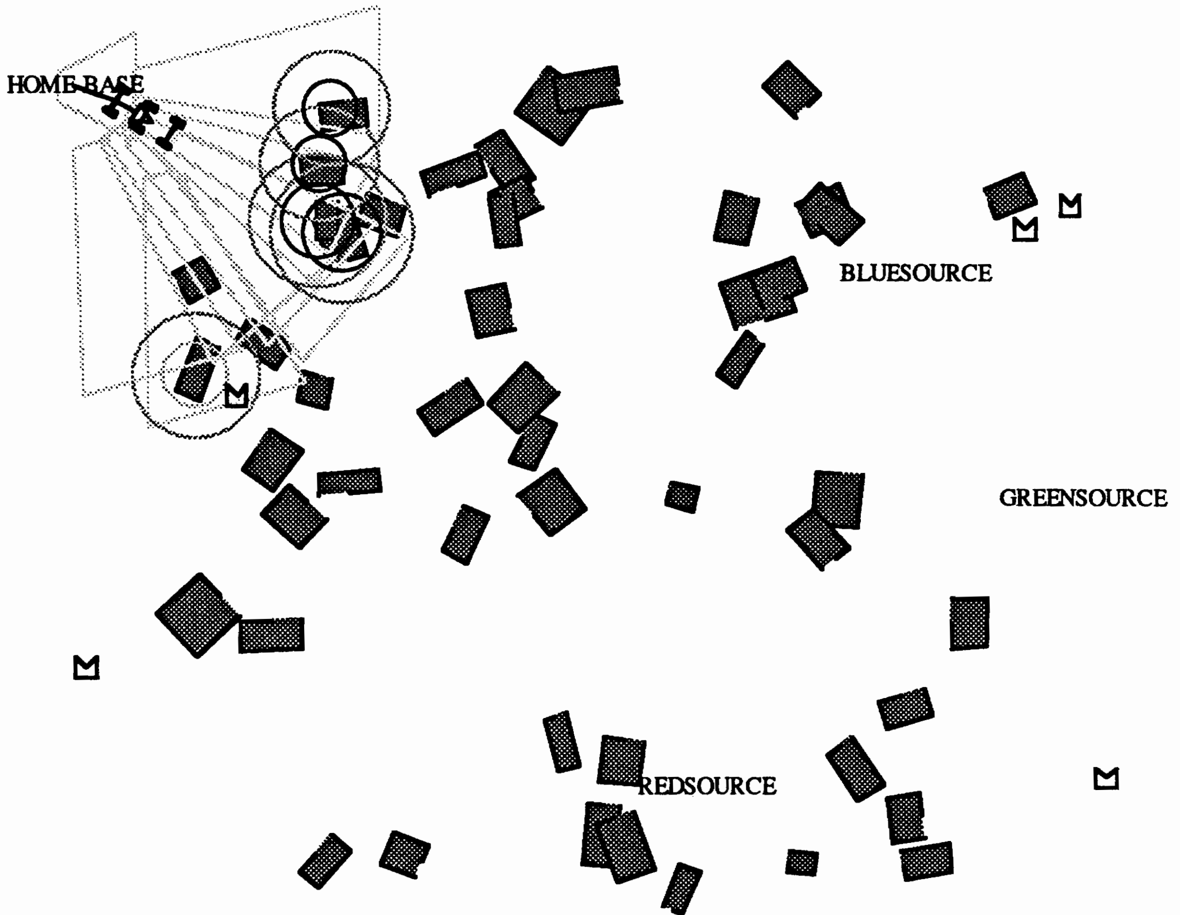


Figure A-2: The robot has just photographed a martian.

```
Beginning cycle 89.  Robot status is :MOVING.
Time is 65124.80
Robot is at x=5.518 y=19.902 h=5.299.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 -7.205442380504579E-4 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 90.  Robot status is :MOVING.
Time is 65139.17
Robot is at x=6.351 y=18.400 h=5.159.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 -0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 91.  Robot status is :MOVING.
Time is 65146.85
Robot is at x=6.612 y=17.907 h=5.330.
```

```
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 -0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 92.  Robot status is :MOVING.
Time is 65156.32
Robot is at x=7.171 y=17.284 h=5.570.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((2.0 -0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 93.  Robot status is :MOVING.
Time is 65165.72
Robot is at x=7.867 y=16.817 h=5.815.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 94.  Robot status is :STOPPED.
Time is 65182.85
Robot is at x=8.196 y=16.672 h=5.885.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 95.  Robot status is :STOPPED.
Time is 65187.28
Robot is at x=8.196 y=16.672 h=5.885.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.2925781430599823.
*** Got a photograph of #<A Martian named DICK>! ***
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.
```

The robot has gotten another martian photograph. The situation at this point is shown in figure A-3.

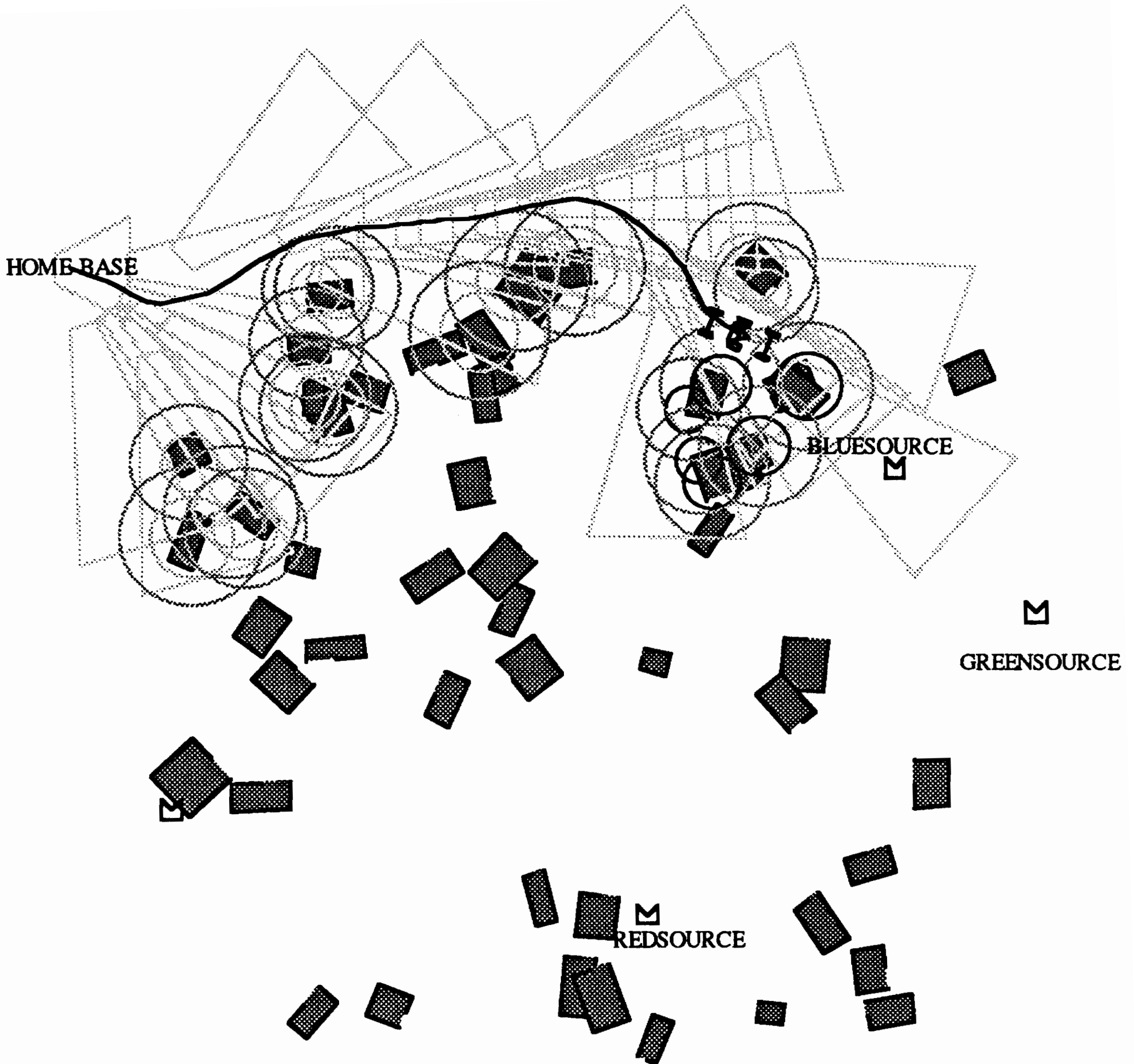


Figure A-3: The robot has photographed another martian.

The robot then continues on its way, squeezing between two closely spaced boulders. Just before it arrives at the blue source, at snaps two martian photographs in close succession. (See figures A-4 and A-5.) These two figures clearly shows that camera being re-aimed in service of martian photographing. The transcript of this section of the journey is uninteresting and has been deleted.

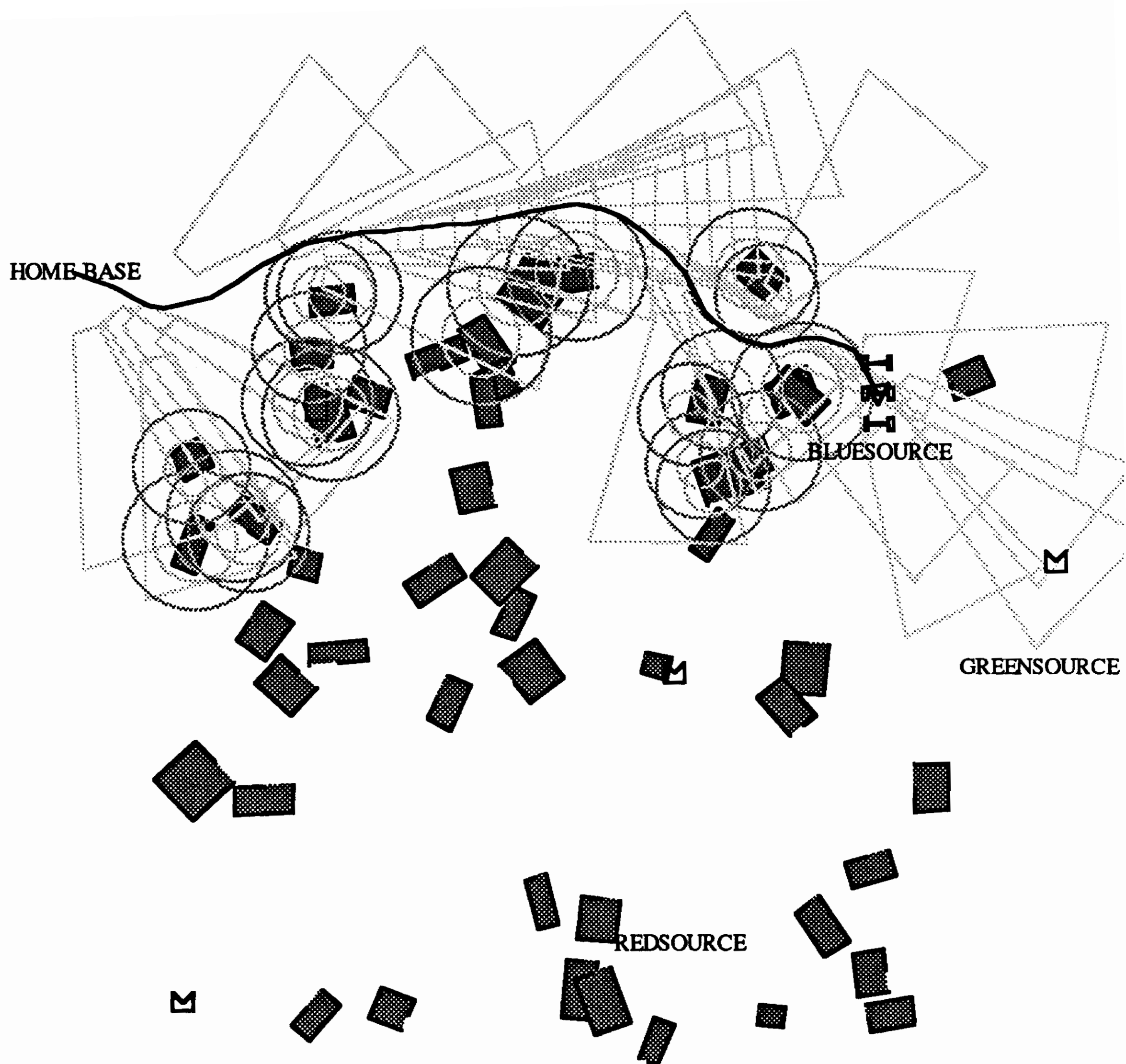


Figure A-4: Another martian is photographed.

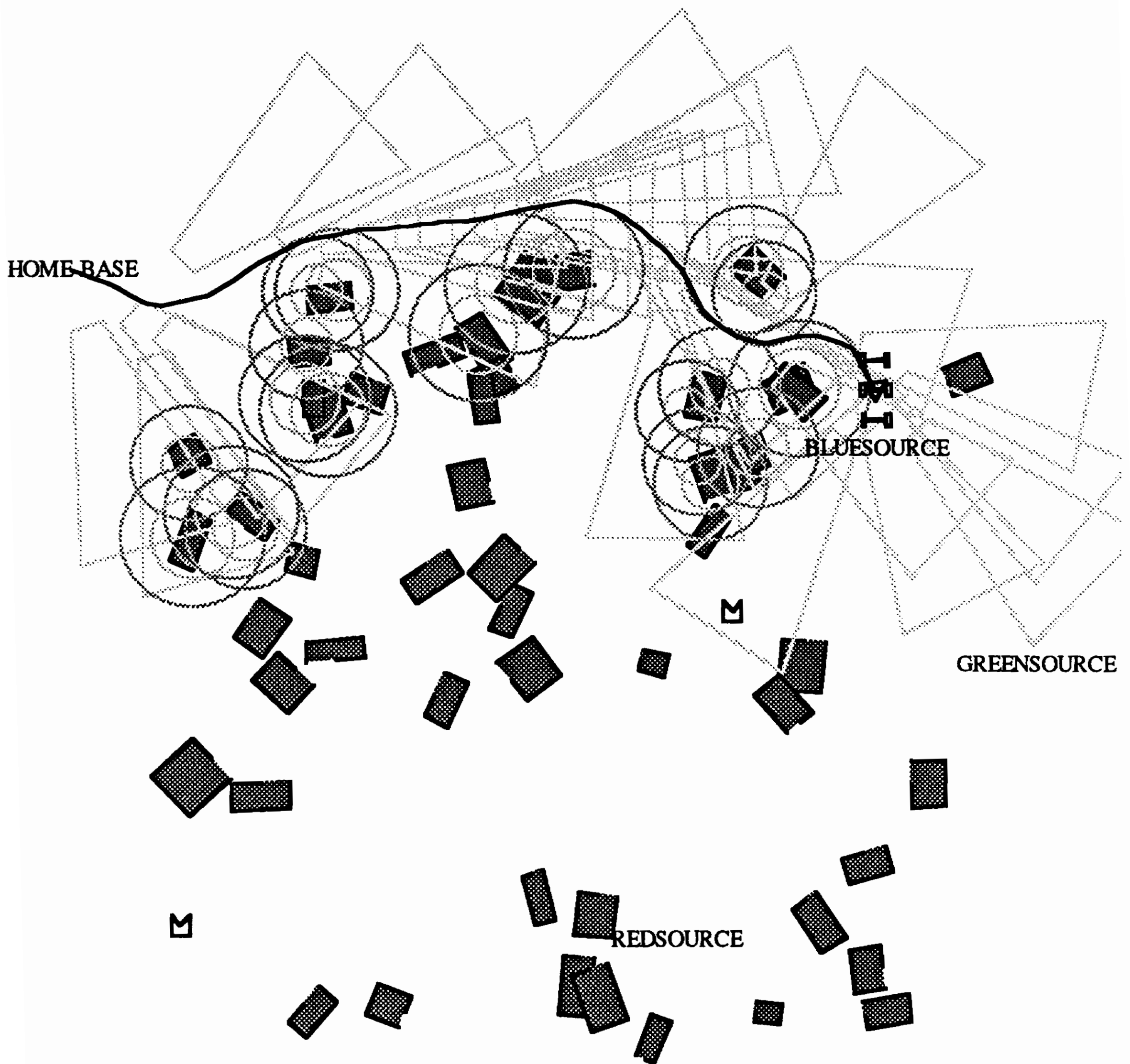


Figure A-5: Four down, one to go.

[34 cycles deleted]

We resume the transcript right before the robot arrives at the blue source.

```
Beginning cycle 129.  Robot status is :STOPPED.
Time is 65472.03
Robot is at x=15.217 y=11.980 h=4.720.
Current plan: (GOTO ROBOT BLUESOURCE)
Moving robot: ((1.003484650326204 0.225929707146074
5.017423251631017))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

```
Beginning cycle 130.  Robot status is :MOVING.
Time is 65478.97
Robot is at x=15.189 y=11.521 h=4.585.
Current plan: (GOTO ROBOT BLUESOURCE)
*** Arrived at BLUESOURCE. ***
Grabbing a frame.  Pan angle is -0.225929707146074.
Cycle complete.  Robot status is: :STOPPED.
```

The situation at this point is essentially identical to that shown in figure A-5, and so a separate figure is not included. The robot goes on to collect the number of rocks it needs.

```
Beginning cycle 131.  Robot status is :STOPPED.
Time is 65507.62
Robot is at x=15.177 y=11.402 h=4.571.
Current plan: (COLLECT-SAMPLE BLUE 5)
*** Collecting 5 BLUE rocks.  ***
New sample status:
Storage available: 5
Samples available: ((BLUE 15) (GREEN 20) (RED 20))
Samples collected: ((BLUE 5) (GREEN 0) (RED 0))
Cycle complete.  Robot status is: :STOPPED.
```

At this point, the robot still has enough fuel and storage space that it does not need to go back to home base right away. Instead, it goes on to collect green rocks. Note that the robot does not have to wait for a new plan. Since the planner has been running continuously in the background, the robot already has a plan ready.

```
Beginning cycle 132.  Robot status is :STOPPED.
Time is 65511.15
Robot is at x=15.177 y=11.402 h=4.571.
Current plan: (GOTO ROBOT GREENSOURCE)
*** Setting goal (23.0 0.0).
Moving robot: ((2.0 -0.625 10.0))
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 133.  Robot status is :MOVING.
Time is 65538.62
Robot is at x=15.195 y=11.143 h=4.811.
Current plan: (GOTO ROBOT GREENSOURCE)
Moving robot: ((2.0 -0.5716638087909587 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.
```

[23 cycles deleted]

```
Beginning cycle 156.  Robot status is :MOVING.
Time is 65732.27
Robot is at x=21.995 y=1.217 h=5.401.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT GREENSOURCE)
```

```
Moving robot: ((1.578184372720242 -1.801381684034986E-3
7.890921863601212))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 157.  Robot status is :MOVING.
Time is 65748.05
Robot is at x=23.290 y=-0.331 h=5.418.
Current plan: (GOTO ROBOT GREENSOURCE)
*** Arrived at GREENSOURCE. ***
Grabbing a frame.  Pan angle is 0.0993416370746596.
Cycle complete.  Robot status is: :STOPPED.
```

The situation now is shown in figure A-6.

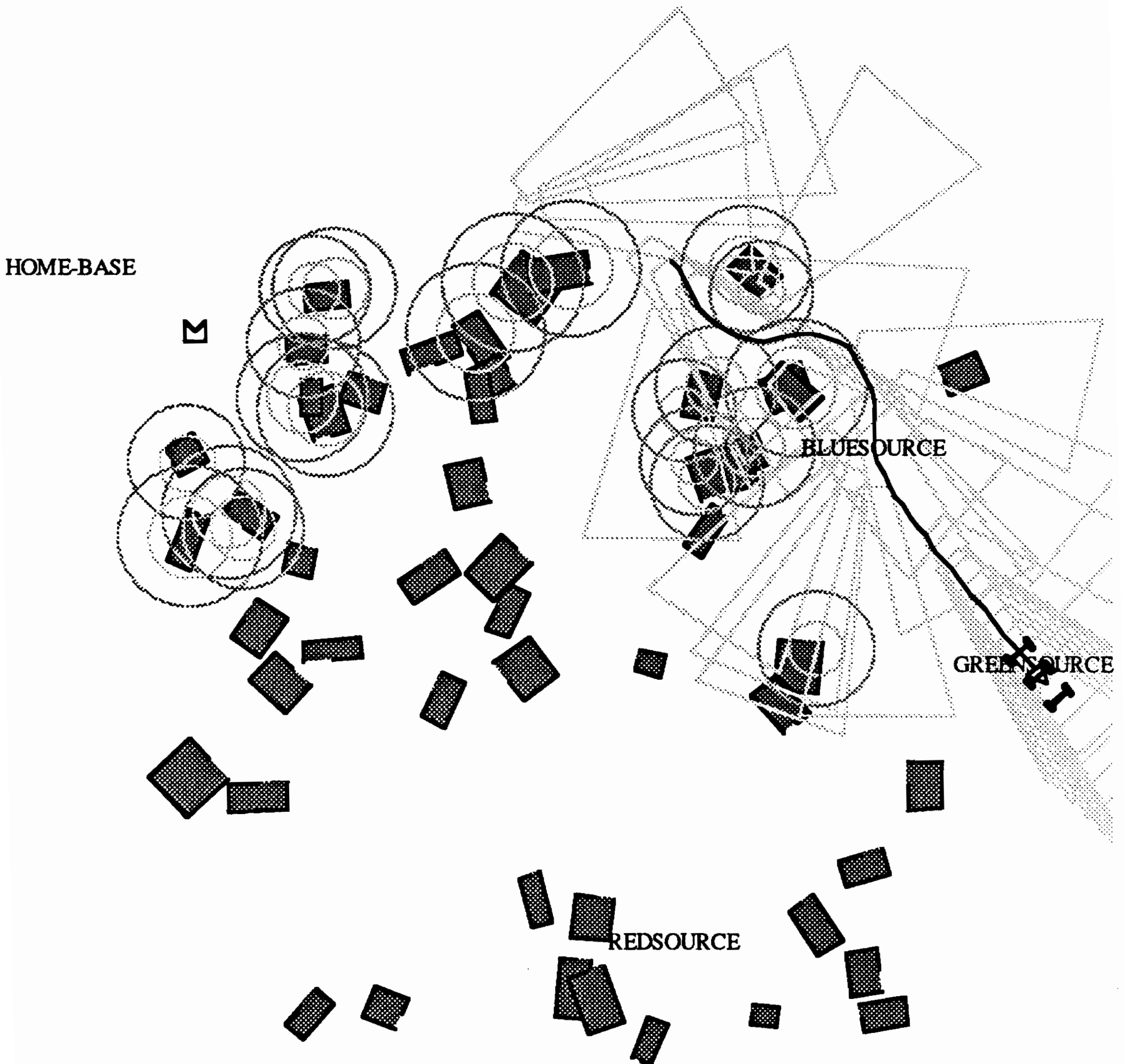


Figure A-6: The robot collects green rocks.

The robot collects its rocks, and goes on to execute the plan it has cached from previous planning, namely, to collect red rocks.

```
Beginning cycle 158.  Robot status is :STOPPED.
Time is 65774.85
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (COLLECT-SAMPLE GREEN 2)
*** Collecting 2 GREEN rocks.  ***
New sample status:
Storage available: 3
Samples available: ((BLUE 15) (GREEN 18) (RED 20))
Samples collected: ((BLUE 5) (GREEN 2) (RED 0))
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 159.  Robot status is :STOPPED.
Time is 65789.23
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Setting goal (5.0 -14.0).
*** Waiting for vision data ***
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 160.  Robot status is :STOPPED.
Time is 65802.45
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 161.  Robot status is :STOPPED.
Time is 65805.53
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 162.  Robot status is :STOPPED.
Time is 65809.32
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 163.  Robot status is :STOPPED.
Time is 65812.08
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.
```

```
Beginning cycle 164.  Robot status is :STOPPED.
Time is 65815.10
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.
```

At this point, the planner realizes that the robot does not have enough fuel to make it to the red source and return home, so it decides to go home first to refuel.

```
Beginning cycle 165.  Robot status is :STOPPED.
Time is 65817.92
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT HOME-BASE)
*** Setting goal (-25.0 20.0).
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 166.  Robot status is :STOPPED.
Time is 65831.75
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 167.  Robot status is :STOPPED.
Time is 65835.17
Robot is at x=23.372 y=-0.419 h=5.424.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.
```

[49 cycles deleted.]

```
Beginning cycle 216.  Robot status is :MOVING.
Time is 66189.25
Robot is at x=8.515 y=4.825 h=3.044.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 8.597155470327689E-2 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.
```

At this point, the robot collides with a boulder that the vision system did not detect due to noise.

```
Beginning cycle 217.  Robot status is :STOPPED.
Time is 66195.83
Robot is at x=7.938 y=4.885 h=3.021.
*** Collision! ***
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.
```

The robot marks the area as impassable, and maneuvers around it. The situation immediately after the collision is shown in figure A-7.

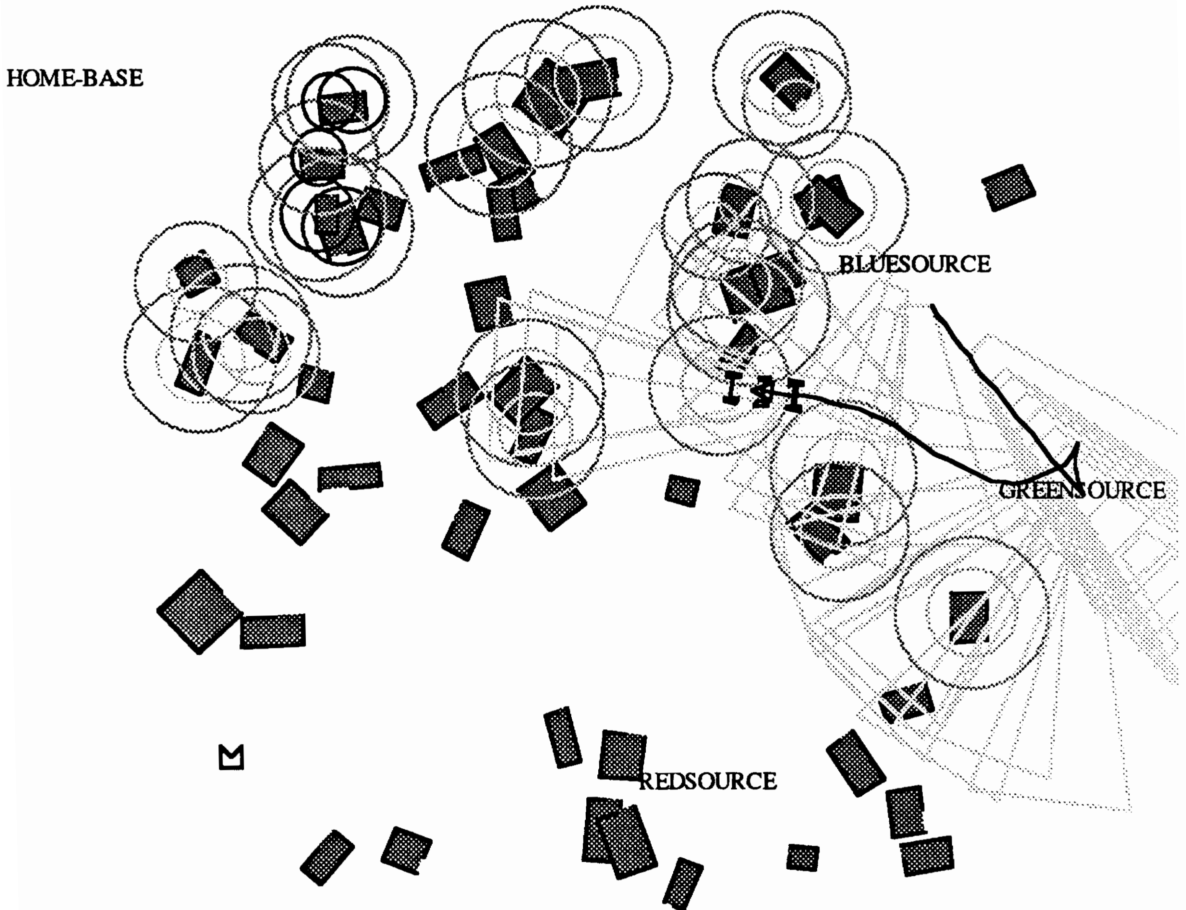


Figure A-7: The robot collides with an unseen boulder.

```
Beginning cycle 218.  Robot status is :STOPPED.
Time is 66224.40
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 219.  Robot status is :STOPPED.
Time is 66238.33
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 220.  Robot status is :STOPPED.
Time is 66241.98
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 221.  Robot status is :STOPPED.
Time is 66245.18
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 222.  Robot status is :STOPPED.
Time is 66248.37
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 223.  Robot status is :STOPPED.
Time is 66252.13
Robot is at x=7.938 y=4.885 h=3.021.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 224.  Robot status is :MOVING.
Time is 66258.52
Robot is at x=8.218 y=4.881 h=3.182.
*** Recovered from collision! ***
```

```
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.0 -0.3125 5.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 225.  Robot status is :MOVING.
Time is 66266.70
Robot is at x=8.855 y=4.918 h=3.310.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.115541722536448 -0.3486067882926399
5.577708612682238))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 226.  Robot status is :MOVING.
Time is 66275.53
Robot is at x=9.589 y=5.103 h=3.473.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 227.  Robot status is :STOPPED.
Time is 66280.37
Robot is at x=9.846 y=5.215 h=3.547.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 228.  Robot status is :STOPPED.
Time is 66283.37
Robot is at x=9.846 y=5.215 h=3.547.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 229.  Robot status is :STOPPED.
Time is 66287.03
Robot is at x=9.846 y=5.215 h=3.547.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 230.  Robot status is :MOVING.
Time is 66293.40
Robot is at x=9.609 y=5.065 h=3.678.
Current plan: (GOTO ROBOT HOME-BASE)
```

```
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 231.  Robot status is :MOVING.
Time is 66312.70
Robot is at x=8.787 y=4.534 h=3.728.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 232.  Robot status is :MOVING.
Time is 66320.50
Robot is at x=8.362 y=4.173 h=3.986.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.2196019466648509 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 233.  Robot status is :MOVING.
Time is 66328.33
Robot is at x=8.004 y=3.743 h=4.132.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.553082142677255 0.2977896187991318
7.765410713386276))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 234.  Robot status is :MOVING.
Time is 66336.15
Robot is at x=7.685 y=3.309 h=3.987.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 235.  Robot status is :MOVING.
Time is 66344.45
Robot is at x=7.633 y=3.166 h=3.822.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.
```

```
Beginning cycle 236.  Robot status is :MOVING.
Time is 66363.43
Robot is at x=8.834 y=4.034 h=3.074.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.187546681334869 -0.3711083379171466
5.937733406674345))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 237.  Robot status is :STOPPED.
Time is 66381.60
Robot is at x=9.690 y=4.184 h=3.383.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 238.  Robot status is :MOVING.
Time is 66388.25
Robot is at x=9.377 y=4.054 h=3.579.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 239.  Robot status is :MOVING.
Time is 66396.40
Robot is at x=8.827 y=3.771 h=3.738.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 240.  Robot status is :MOVING.
Time is 66404.57
Robot is at x=8.339 y=3.393 h=3.947.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.994985771266893 0.6234330535209041
9.974928856334465))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 241.  Robot status is :MOVING.
Time is 66412.45
Robot is at x=8.326 y=3.348 h=3.820.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.777231707538476 0.5553849086057738
8.886158537692381))
```

```
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 242.  Robot status is :MOVING.
Time is 66420.75
Robot is at x=8.876 y=3.710 h=3.635.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 243.  Robot status is :MOVING.
Time is 66428.98
Robot is at x=8.815 y=3.655 h=3.657.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 -0.3125 5.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.5383881210804007.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 244.  Robot status is :MOVING.
Time is 66437.02
Robot is at x=8.348 y=3.313 h=3.888.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 245.  Robot status is :STOPPED.
Time is 66456.23
Robot is at x=9.924 y=3.931 h=3.365.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.044150637958957 -0.3262970743621739
5.220753189794783))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 246.  Robot status is :MOVING.
Time is 66463.12
Robot is at x=9.600 y=3.829 h=3.526.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.065856737900533 -0.3330802305939166
5.329283689502666))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 247.  Robot status is :MOVING.
Time is 66471.37
```

```
Robot is at x=9.009 y=3.541 h=3.752.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.37994361314448 -0.2138240262515443
6.899718065722401))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 248.  Robot status is :MOVING.
Time is 66479.68
Robot is at x=8.487 y=3.171 h=3.758.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 249.  Robot status is :MOVING.
Time is 66487.95
Robot is at x=8.520 y=3.195 h=3.776.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -3.150386126655569E-2.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 250.  Robot status is :MOVING.
Time is 66496.08
Robot is at x=9.033 y=3.505 h=3.590.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.119903961098646 -0.3499699878433268
5.599519805493229))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 251.  Robot status is :MOVING.
Time is 66505.00
Robot is at x=8.909 y=3.438 h=3.681.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.330219889526984 -0.2813231250682278
6.651099447634918))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 252.  Robot status is :MOVING.
Time is 66513.40
Robot is at x=8.363 y=3.070 h=3.789.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.4389400822257059 10.0))
Aiming camera.
Planning.
Time to refuel.
```

```
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 253.  Robot status is :MOVING.
Time is 66533.50
Robot is at x=7.424 y=2.094 h=3.869.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 254.  Robot status is :MOVING.
Time is 66541.95
Robot is at x=7.472 y=2.062 h=3.666.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 255.  Robot status is :MOVING.
Time is 66550.25
Robot is at x=7.481 y=2.150 h=3.357.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 256.  Robot status is :MOVING.
Time is 66559.17
Robot is at x=6.708 y=2.062 h=3.138.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.5212095608435945 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 257.  Robot status is :MOVING.
Time is 66567.23
Robot is at x=6.111 y=2.108 h=2.986.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.2705218661391697 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 258.  Robot status is :MOVING.
Time is 66575.12
Robot is at x=5.543 y=2.223 h=2.906.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.3656120869011001 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
```

```
Pan angle is -0.2705218661391697.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 259.  Robot status is :MOVING.
Time is 66583.47
Robot is at x=4.930 y=2.406 h=2.792.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.473072185654l695 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 260.  Robot status is :STOPPED.
Time is 66602.95
Robot is at x=3.046 y=3.717 h=2.437.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.1142653211094693 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 261.  Robot status is :MOVING.
Time is 66609.35
Robot is at x=2.852 y=3.890 h=2.409.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.0799481595066922 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.1142653211094693.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 262.  Robot status is :MOVING.
Time is 66617.80
Robot is at x=2.352 y=4.351 h=2.383.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -9.699864841455774E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 263.  Robot status is :MOVING.
Time is 66625.02
Robot is at x=1.836 y=4.824 h=2.417.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -1.960162353186012E-2 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 264.  Robot status is :MOVING.
Time is 66633.02
Robot is at x=1.387 y=5.222 h=2.416.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.185949272851925 10.0))
Aiming camera.
Planning.
```

```
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 265.  Robot status is :MOVING.
Time is 66640.85
Robot is at x=0.977 y=5.603 h=2.358.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 5.514948161578204E-2 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 266.  Robot status is :MOVING.
Time is 66648.65
Robot is at x=0.569 y=6.015 h=2.345.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.0576102603634463 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 267.  Robot status is :MOVING.
Time is 66656.43
Robot is at x=0.175 y=6.413 h=2.364.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1568602964319963 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.0576102603634463.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 268.  Robot status is :STOPPED.
Time is 66676.73
Robot is at x=-1.562 y=7.917 h=2.442.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.625 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 269.  Robot status is :MOVING.
Time is 66684.13
Robot is at x=-1.978 y=8.156 h=2.666.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.407604081980276 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 270.  Robot status is :MOVING.
Time is 66692.02
Robot is at x=-2.509 y=8.390 h=2.794.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
```

```
Planning.
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 271.  Robot status is :STOPPED.
Time is 66696.58
Robot is at x=-2.736 y=8.466 h=2.842.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.0.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 272.  Robot status is :STOPPED.
Time is 66699.70
Robot is at x=-2.736 y=8.466 h=2.842.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 273.  Robot status is :STOPPED.
Time is 66703.85
Robot is at x=-2.736 y=8.466 h=2.842.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.2696180769209993 10.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.
```

At this point the robot has another collision with an unseen boulder. (See figure A-8.) Note that the new information gained from this collision requires a major change in the robot's route.

```
Beginning cycle 274.  Robot status is :STOPPED.
Time is 66708.67
Robot is at x=-2.965 y=8.539 h=2.855.
*** Collision! ***
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.0 0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.
```

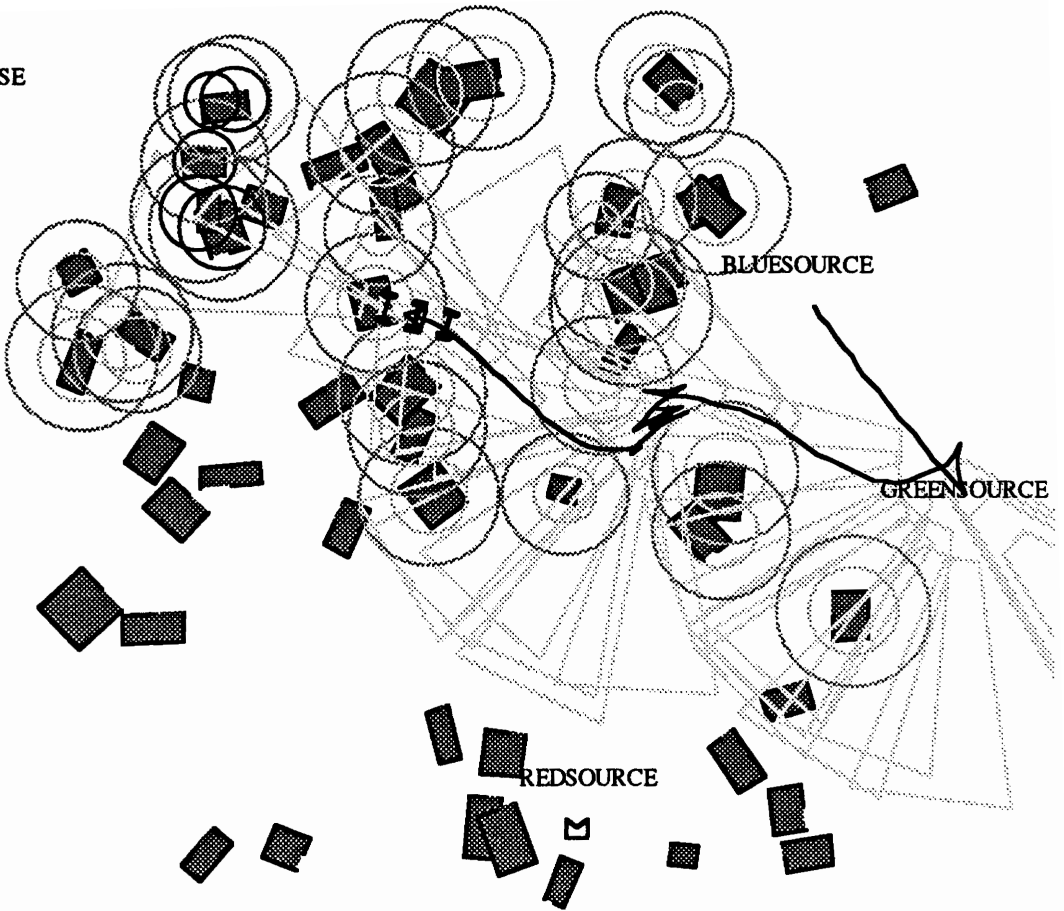


Figure A-8: A second collision with an unseen boulder.

```
Beginning cycle 275.  Robot status is :MOVING.
Time is 66745.87
Robot is at x=-2.721 y=8.448 h=2.718.
*** Recovered from collision! ***
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.0 0.3125 5.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 276.  Robot status is :MOVING.
Time is 66765.27
Robot is at x=-2.140 y=8.140 h=2.493.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.022758083813476 0.3196119011917111
5.113790419067377))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.
```

```
Beginning cycle 277.  Robot status is :MOVING.
Time is 66774.65
Robot is at x=-1.532 y=7.567 h=2.198.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.0 0.3125 5.0))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 278.  Robot status is :MOVING.
Time is 66783.13
Robot is at x=-1.395 y=7.460 h=1.925.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-1.209299687172755 -0.377906152241486
6.046498435863776))
Aiming camera.
Planning.
Time to refuel.
Cycle complete.  Robot status is: :MOVING.
```

[Twelve cycles deleted]

At this point, the robot realizes that it might not have enough fuel to make it back to the home base. This is because the return trip is taking longer than expected due to the two unexpected collisions.

```
Beginning cycle 290.  Robot status is :MOVING.
Time is 66872.20
Robot is at x=-1.421 y=8.952 h=2.230.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Warning!  In danger of running out of fuel!
Cycle complete.  Robot status is: :STOPPED.
```

[76 cycles deleted]

At this point the robot actually ran out of fuel. The simulation was allowed to continue on "reserve fuel", and thirty-three seconds later the robot arrived at the home base and refueled. (See figure A-9.) In over twenty hours of operation, this is the only recorded instance of the robot running out of fuel.

```
Beginning cycle 366.  Robot status is :STOPPED.
Time is 67515.43
Robot is at x=-21.783 y=20.583 h=3.328.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 7.382830754533387E-3 10.0))
Aiming camera.
Planning.
Warning!  In danger of running out of fuel!
Cycle complete.  Robot status is: :MOVING.
```

```
Beginning cycle 367.  Robot status is :MOVING.
Time is 67520.57
Robot is at x=-21.901 y=20.562 h=3.319.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Robot is out of fuel!
Warning!  In danger of running out of fuel!
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 368.  Robot status is :STOPPED.
Time is 67524.37
Robot is at x=-22.118 y=20.524 h=3.317.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -4.854446303899973E-3 10.0))
Aiming camera.
Planning.
Robot is out of fuel!
Warning!  In danger of running out of fuel!
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 369.  Robot status is :MOVING.
Time is 67529.62
Robot is at x=-22.256 y=20.500 h=3.313.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -8.348532413360754E-3 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 4.854446303899973E-3.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 370.  Robot status is :MOVING.
Time is 67537.20
Robot is at x=-22.788 y=20.407 h=3.316.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -6.942216596741879E-3 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 371.  Robot status is :MOVING.
Time is 67546.85
Robot is at x=-23.674 y=20.249 h=3.320.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.349413759333955 -7.326058882072939E-3
6.747068796669777))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 4.854446303899973E-3.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 372.  Robot status is :MOVING.
Time is 67553.07
Robot is at x=-24.185 y=20.156 h=3.325.
Current plan: (GOTO ROBOT HOME-BASE)
```

```
*** Arrived at HOME-BASE. ***
Robot is out of fuel!
Time to refuel.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 373.  Robot status is :STOPPED.
Time is 67604.98
Robot is at x=-24.303 y=20.134 h=3.338.
Telemetry: :BOULDER-DATA
Current plan: (REFUEL)
*** Refueling robot. ***
Cycle complete.  Robot status is: :STOPPED.
```

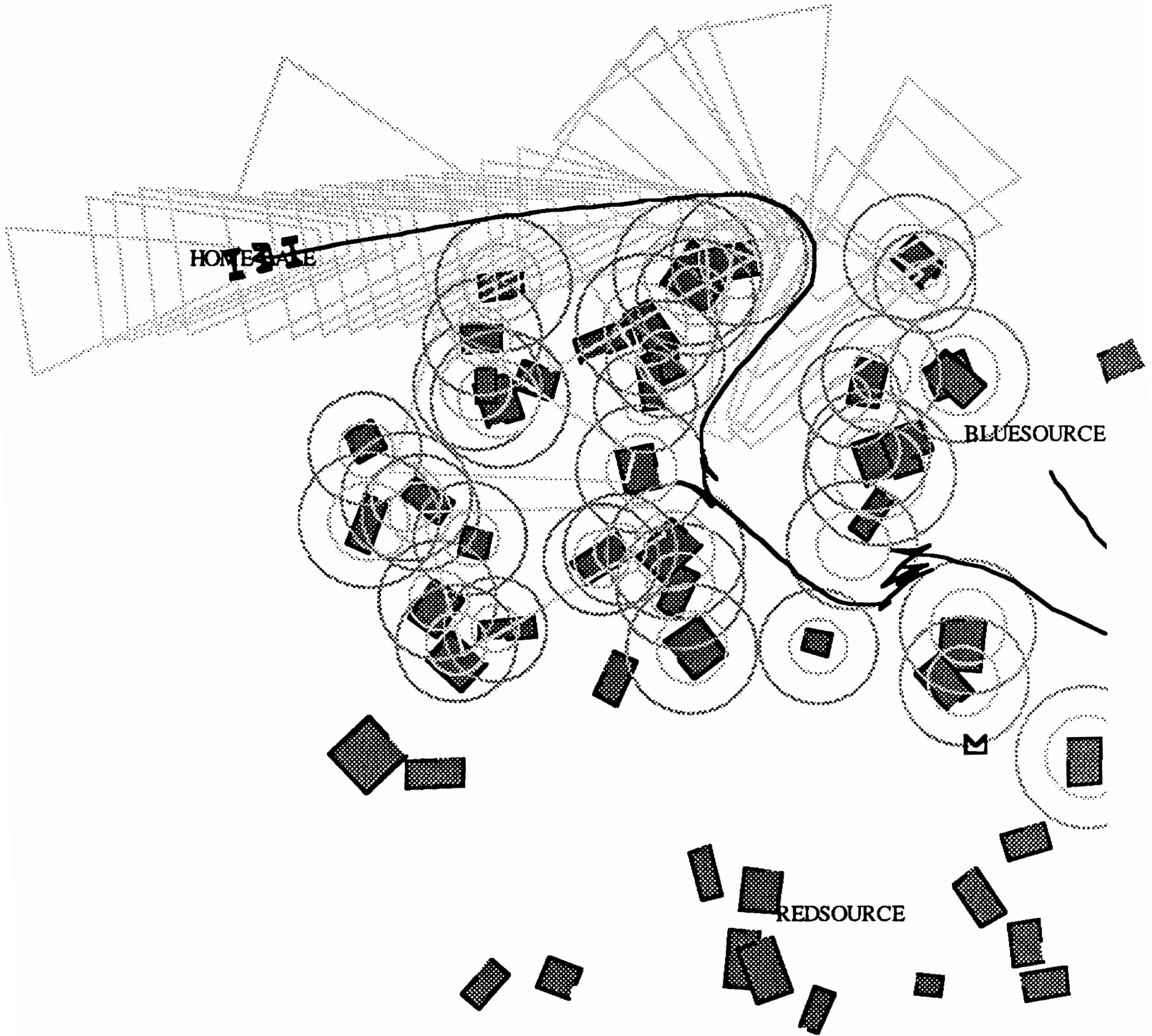


Figure A-9: The robot arrives at home base, refuels, and delivers blue and green rocks.

The robot delivers the rocks it has collected, and goes on to its final task of getting red rocks.

```
Beginning cycle 374.  Robot status is :STOPPED.
Time is 67621.15
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (DELIVER-SAMPLE GREEN 2)
*** Delivering 2 GREEN rocks. ***
New sample status:
Storage available: 5
Samples available: ((BLUE 15) (GREEN 18) (RED 20))
Samples collected: ((BLUE 5) (GREEN 0) (RED 0))
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 375.  Robot status is :STOPPED.
Time is 67623.68
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (DELIVER-SAMPLE BLUE 5)
*** Delivering 5 BLUE rocks. ***
New sample status:
Storage available: 10
Samples available: ((BLUE 15) (GREEN 18) (RED 20))
Samples collected: ((BLUE 0) (GREEN 0) (RED 0))
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 376.  Robot status is :STOPPED.
Time is 67626.70
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Setting goal (5.0 -14.0).
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 377.  Robot status is :STOPPED.
Time is 67645.78
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 378.  Robot status is :STOPPED.
Time is 67660.40
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 379.  Robot status is :STOPPED.
Time is 67663.47
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
```

```
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 380.  Robot status is :STOPPED.
Time is 67668.33
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 381.  Robot status is :STOPPED.
Time is 67672.23
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 382.  Robot status is :STOPPED.
Time is 67675.45
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 383.  Robot status is :STOPPED.
Time is 67680.50
Robot is at x=-24.303 y=20.134 h=3.338.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((-2.0 -0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

[40 cycles deleted.]

```
Beginning cycle 423.  Robot status is :MOVING.
Time is 68014.68
Robot is at x=-11.377 y=7.327 h=5.435.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 424.  Robot status is :MOVING.
Time is 68023.17
Robot is at x=-10.981 y=6.776 h=5.228.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 0.5445017730927741 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

```
Beginning cycle 425.  Robot status is :MOVING.
Time is 68031.53
Robot is at x=-10.703 y=6.157 h=5.040.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 0.236227769544711 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 426.  Robot status is :MOVING.
Time is 68039.72
Robot is at x=-10.521 y=5.543 h=4.964.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 1.507298125079881E-2 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 427.  Robot status is :MOVING.
Time is 68047.95
Robot is at x=-10.365 y=4.923 h=4.959.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -7.638228538943359E-3 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

At this point, the robot gets a picture of the last martian. (See figure A-10.)

```
Beginning cycle 428.  Robot status is :MOVING.
Time is 68055.95
Robot is at x=-10.218 y=4.341 h=4.965.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -0.2299615170661085 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 1.309660119068282.
*** Got a photograph of #<A Martian named FRED>! ***
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.
```

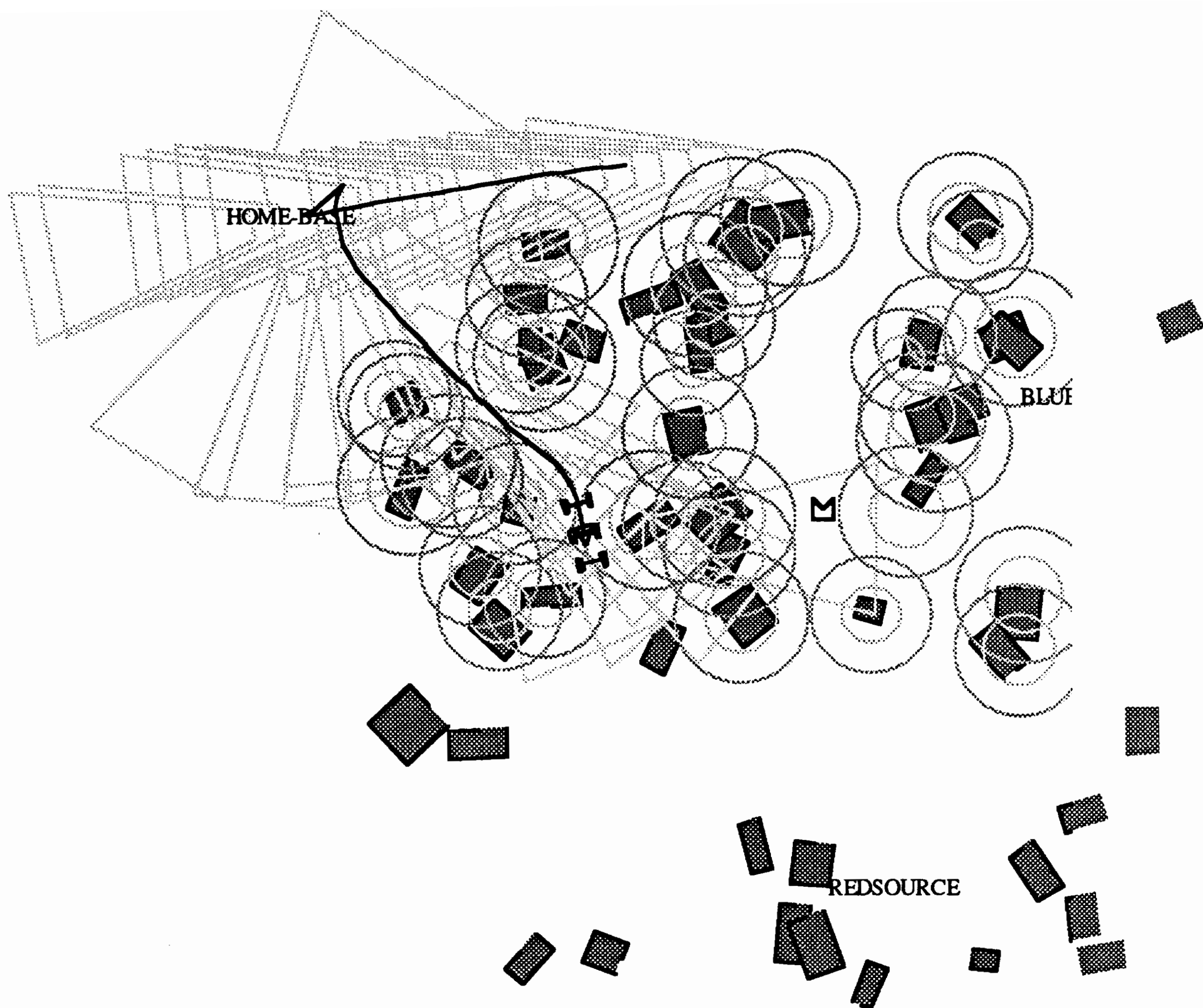


Figure A-10: The last martian is photographed.

```
Beginning cycle 429.  Robot status is :STOPPED.
Time is 68107.20
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 430.  Robot status is :STOPPED.
Time is 68111.97
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 431.  Robot status is :STOPPED.
```

```
Time is 68115.08
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.0.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 432.  Robot status is :STOPPED.
Time is 68117.90
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 433.  Robot status is :STOPPED.
Time is 68122.55
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 434.  Robot status is :STOPPED.
Time is 68125.65
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is 0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 435.  Robot status is :STOPPED.
Time is 68128.45
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 436.  Robot status is :STOPPED.
Time is 68133.08
Robot is at x=-9.459 y=2.216 h=5.079.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -0.3875426511221978 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 437.  Robot status is :MOVING.
Time is 68139.20
Robot is at x=-9.324 y=1.948 h=5.196.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -0.2666953893479542 10.0))
```

```
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 438.  Robot status is :MOVING.
Time is 68159.88
Robot is at x=-8.858 y=1.178 h=5.314.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -0.1495684828914658 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 439.  Robot status is :MOVING.
Time is 68167.78
Robot is at x=-8.522 y=0.706 h=5.348.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -0.1168289212817433 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.1495684828914658.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 440.  Robot status is :MOVING.
Time is 68175.57
Robot is at x=-8.194 y=0.276 h=5.377.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 -8.844227503192315E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.
```

At this point, the robot collides with a third unseen boulder. (See figure A-11.)

```
Beginning cycle 441.  Robot status is :STOPPED.
Time is 68183.68
Robot is at x=-7.651 y=-0.390 h=5.417.
*** Collision! ***
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.
```

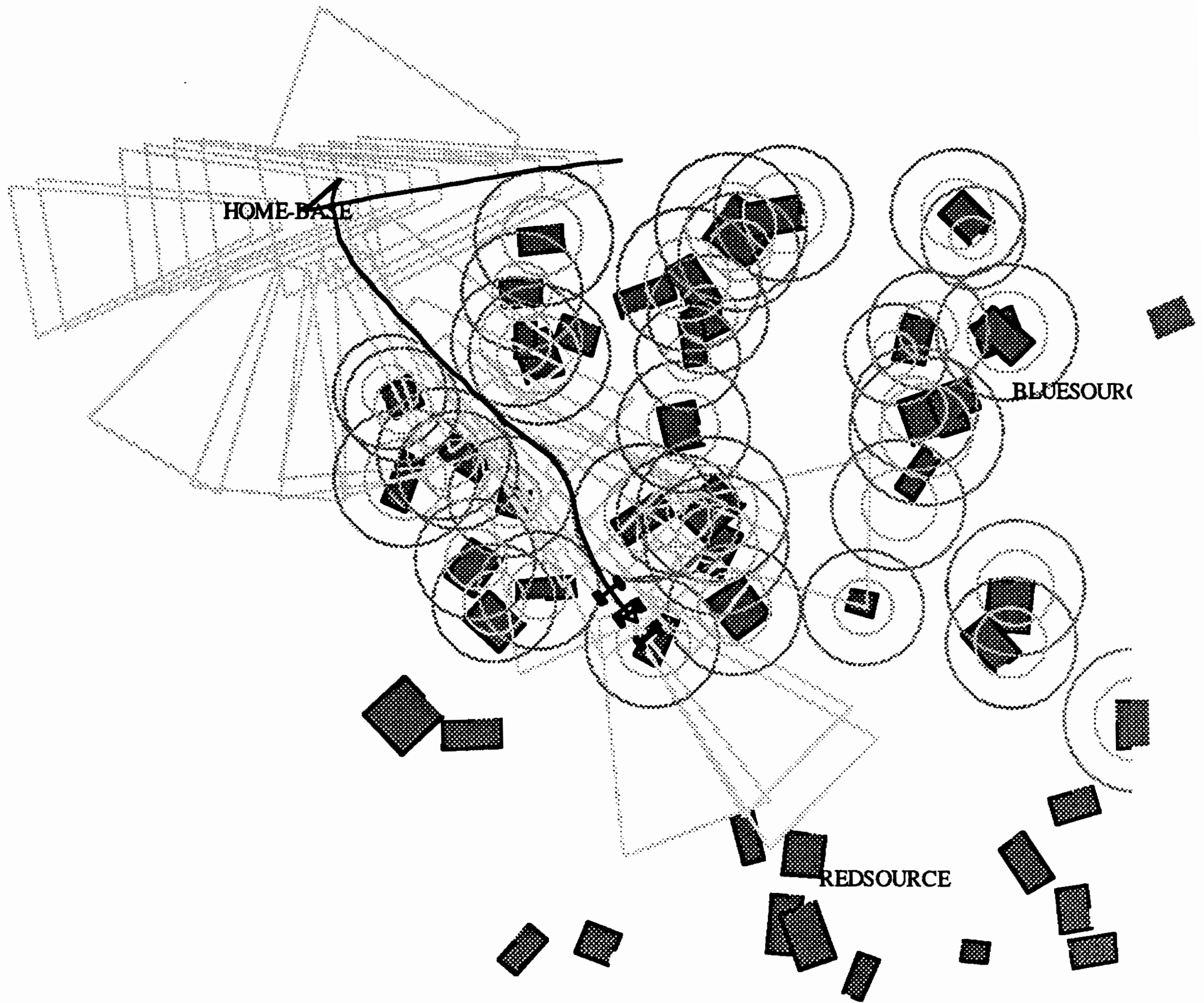


Figure A-11: The third (and last) collision.

```
Beginning cycle 442.  Robot status is :STOPPED.
Time is 68219.68
Robot is at x=-7.651 y=-0.390 h=5.417.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 443.  Robot status is :STOPPED.
Time is 68246.25
Robot is at x=-7.651 y=-0.390 h=5.417.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
```

```
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 444.  Robot status is :STOPPED.
Time is 68251.57
Robot is at x=-7.651 y=-0.390 h=5.417.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 445.  Robot status is :STOPPED.
Time is 68255.38
Robot is at x=-7.651 y=-0.390 h=5.417.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 446.  Robot status is :STOPPED.
Time is 68258.58
Robot is at x=-7.651 y=-0.390 h=5.417.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 447.  Robot status is :STOPPED.
Time is 68263.75
Robot is at x=-7.651 y=-0.390 h=5.417.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((-1.0 0.3125 5.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 448.  Robot status is :STOPPED.
Time is 68269.33
Robot is at x=-7.831 y=-0.126 h=5.316.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 449.  Robot status is :STOPPED.
Time is 68272.85
Robot is at x=-7.831 y=-0.126 h=5.316.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 450.  Robot status is :STOPPED.
Time is 68276.08
Robot is at x=-7.831 y=-0.126 h=5.316.
```

```
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 451.  Robot status is :STOPPED.
Time is 68281.30
Robot is at x=-7.831 y=-0.126 h=5.316.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((-1.0 0.3125 5.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 452.  Robot status is :MOVING.
Time is 68288.37
Robot is at x=-7.995 y=0.172 h=5.141.
*** Recovered from collision! ***
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((-1.0 0.3125 5.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.
```

At this point, the local path planner cannot find a path because there are too many obstacles nearby. The planner punts and generates a skid turn. This is the only skid turn in the transcript.

```
Beginning cycle 453.  Robot status is :MOVING.
Time is 68307.78
Robot is at x=-8.242 y=0.803 h=5.018.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((0.0 0.1 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 454.  Robot status is :STOPPED.
Time is 68315.30
Robot is at x=-8.242 y=0.803 h=4.918.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((0.0 0.1 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 455.  Robot status is :STOPPED.
Time is 68320.13
Robot is at x=-8.242 y=0.803 h=4.818.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.
```

```
Beginning cycle 456.  Robot status is :STOPPED.
Time is 68326.23
Robot is at x=-8.242 y=0.803 h=4.818.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 457.  Robot status is :STOPPED.
Time is 68330.08
Robot is at x=-8.242 y=0.803 h=4.818.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 458.  Robot status is :STOPPED.
Time is 68333.40
Robot is at x=-8.242 y=0.803 h=4.818.
Current plan: (GOTO ROBOT REDSOURCE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 459.  Robot status is :STOPPED.
Time is 68339.08
Robot is at x=-8.242 y=0.803 h=4.818.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((1.0 0.3125 5.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 460.  Robot status is :MOVING.
Time is 68346.23
Robot is at x=-8.244 y=0.463 h=4.726.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((1.0 0.3125 5.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 461.  Robot status is :MOVING.
Time is 68354.65
Robot is at x=-8.324 y=-0.210 h=4.450.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((1.0 0.3125 5.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.
```

[37 cycles deleted.]

```
Beginning cycle 498.  Robot status is :MOVING.
```

```
Time is 68689.32
Robot is at x=4.692 y=-11.958 h=5.112.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((2.0 0.250007659915414 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.2633476254028482.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 499.  Robot status is :MOVING.
Time is 68697.15
Robot is at x=4.886 y=-12.462 h=5.045.
Current plan: (GOTO ROBOT REDSOURCE)
Moving robot: ((1.542601909996361 0.2586851917363724
7.713009549981804))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.
```

The robot arrives at the red source without further difficulty. (See figure A-12.)

```
Beginning cycle 500.  Robot status is :MOVING.
Time is 68708.58
Robot is at x=5.170 y=-13.645 h=4.817.
Current plan: (GOTO ROBOT REDSOURCE)
*** Arrived at REDSOURCE. ***
Grabbing a frame.  Pan angle is -0.2633476254028482.
Cycle complete.  Robot status is: :STOPPED.
```

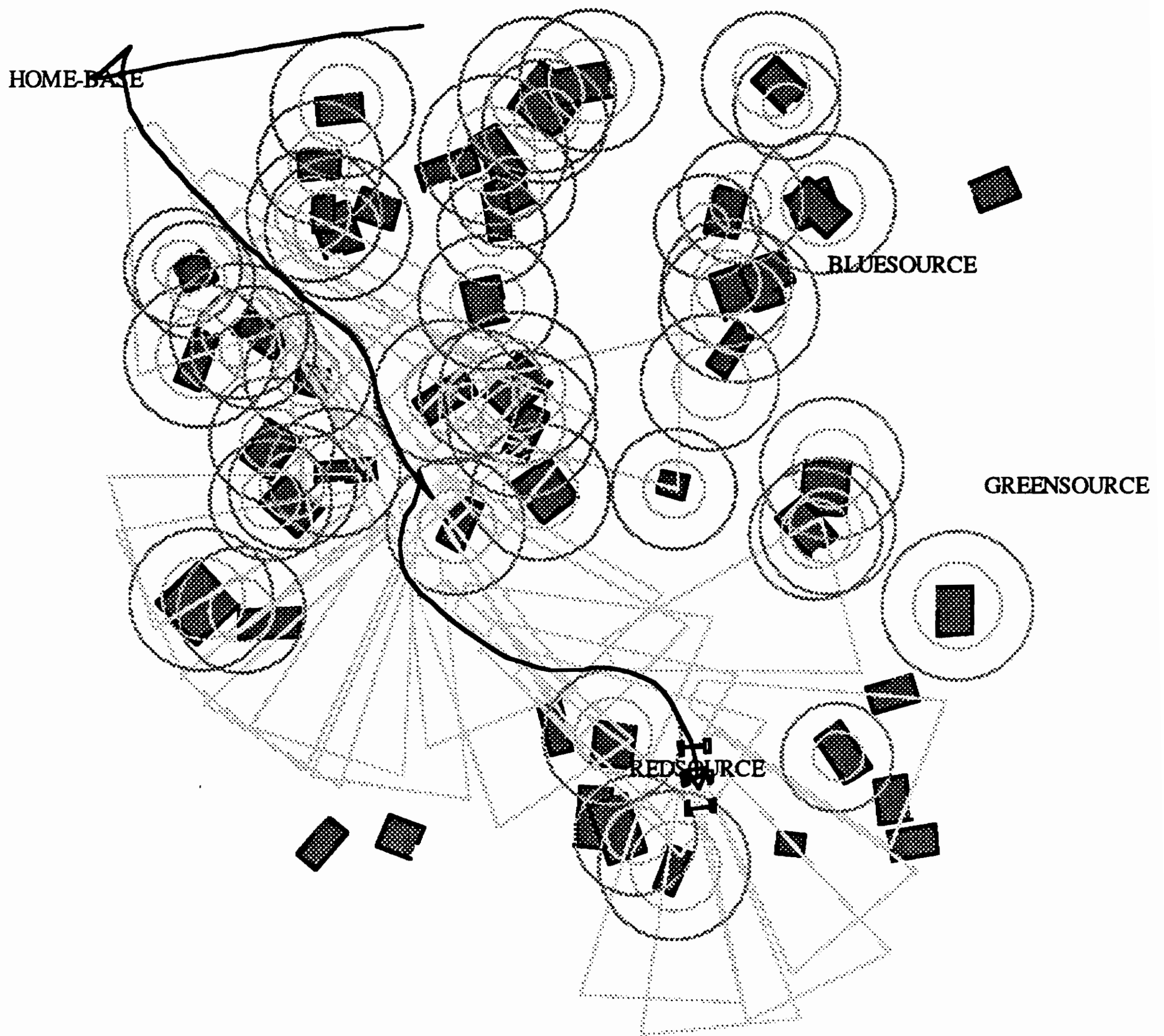


Figure A-12: The robot arrives at the red source.

```
Beginning cycle 501.  Robot status is :STOPPED.
Time is 68757.40
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (COLLECT-SAMPLE RED 3)
*** Collecting 3 RED rocks.  ***
New sample status:
Storage available: 7
Samples available: ((BLUE 15) (GREEN 18) (RED 17))
Samples collected: ((BLUE 0) (GREEN 0) (RED 3))
Cycle complete.  Robot status is: :STOPPED.
```

After this, the robot returns home and delivers the red rocks without further incident. The rest of the transcript is therefore highly uninteresting. It is left intact to give a flavor for what the normal behavior of the system looks like.

```
Beginning cycle 502.  Robot status is :STOPPED.
Time is 68760.23
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Setting goal (-25.0 20.0).
*** Waiting for vision data ***
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 503.  Robot status is :STOPPED.
Time is 68780.30
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 504.  Robot status is :STOPPED.
Time is 68786.85
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 505.  Robot status is :STOPPED.
Time is 68790.35
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 506.  Robot status is :STOPPED.
Time is 68804.53
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 507.  Robot status is :STOPPED.
Time is 68810.95
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 508.  Robot status is :STOPPED.
Time is 68814.55
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
```

```
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 509.  Robot status is :STOPPED.
Time is 68817.85
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 510.  Robot status is :STOPPED.
Time is 68824.88
Robot is at x=5.231 y=-14.246 h=4.808.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 511.  Robot status is :MOVING.
Time is 68831.90
Robot is at x=5.264 y=-13.907 h=4.583.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 512.  Robot status is :STOPPED.
Time is 68837.48
Robot is at x=5.322 y=-13.532 h=4.541.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 513.  Robot status is :STOPPED.
Time is 68841.10
Robot is at x=5.322 y=-13.532 h=4.541.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 514.  Robot status is :STOPPED.
Time is 68848.65
Robot is at x=5.322 y=-13.532 h=4.541.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 515.  Robot status is :MOVING.
```

```
Time is 68881.28
Robot is at x=5.420 y=-13.206 h=4.381.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -1.5.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 516.  Robot status is :MOVING.
Time is 68889.97
Robot is at x=5.713 y=-12.573 h=4.170.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 517.  Robot status is :MOVING.
Time is 68902.78
Robot is at x=6.715 y=-11.527 h=3.691.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((-2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 518.  Robot status is :MOVING.
Time is 68911.80
Robot is at x=7.411 y=-11.181 h=3.465.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 519.  Robot status is :STOPPED.
Time is 68931.12
Robot is at x=5.813 y=-10.854 h=3.020.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 520.  Robot status is :MOVING.
Time is 68938.05
Robot is at x=5.488 y=-10.758 h=2.822.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.4650276470414845 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 521.  Robot status is :MOVING.
Time is 68946.92
Robot is at x=4.770 y=-10.454 h=2.653.
Current plan: (GOTO ROBOT HOME-BASE)
```

```
Moving robot: ((2.0 0.1589003689232085 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 522.  Robot status is :MOVING.
Time is 68955.85
Robot is at x=4.115 y=-10.069 h=2.586.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1733206290435358 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 523.  Robot status is :MOVING.
Time is 68964.32
Robot is at x=3.517 y=-9.706 h=2.630.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1334111665223428 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.1733206290435358.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 524.  Robot status is :MOVING.
Time is 68972.78
Robot is at x=2.936 y=-9.393 h=2.671.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -9.398038637515249E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 525.  Robot status is :MOVING.
Time is 68996.27
Robot is at x=1.669 y=-8.798 h=2.692.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -8.044340285421514E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.1733206290435358.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 526.  Robot status is :MOVING.
Time is 69004.95
Robot is at x=1.008 y=-8.515 h=2.734.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -3.260289911918512E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 527.  Robot status is :MOVING.
Time is 69017.35
Robot is at x=-0.267 y=-7.987 h=2.802.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 4.676076172485999E-2 10.0))
Aiming camera.
```

```
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 528.  Robot status is :MOVING.
Time is 69026.07
Robot is at x=-0.956 y=-7.717 h=2.773.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 1.139793820811974E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -4.676076172485999E-2.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 529.  Robot status is :MOVING.
Time is 69034.58
Robot is at x=-1.572 y=-7.481 h=2.775.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.0167868306279324 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 530.  Robot status is :MOVING.
Time is 69046.97
Robot is at x=-2.878 y=-6.975 h=2.810.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 7.136147260700554E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -4.676076172485999E-2.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 531.  Robot status is :MOVING.
Time is 69066.72
Robot is at x=-3.549 y=-6.715 h=2.775.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 2.648994420590878E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 532.  Robot status is :MOVING.
Time is 69079.42
Robot is at x=-4.889 y=-6.190 h=2.802.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 9.728911264603113E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -4.676076172485999E-2.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 533.  Robot status is :MOVING.
Time is 69088.73
Robot is at x=-5.644 y=-5.871 h=2.707.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 8.901792867759184E-2 10.0))
Boulder processing a vision frame ... done.
```

```
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 534.  Robot status is :MOVING.
Time is 69101.58
Robot is at x=-6.929 y=-5.221 h=2.600.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 535.  Robot status is :STOPPED.
Time is 69107.08
Robot is at x=-7.243 y=-5.045 h=2.584.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 536.  Robot status is :STOPPED.
Time is 69110.57
Robot is at x=-7.243 y=-5.045 h=2.584.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 537.  Robot status is :STOPPED.
Time is 69117.75
Robot is at x=-7.243 y=-5.045 h=2.584.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.3601972302135206 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 538.  Robot status is :MOVING.
Time is 69135.77
Robot is at x=-7.503 y=-4.860 h=2.504.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.3458433833089387 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 539.  Robot status is :MOVING.
Time is 69148.27
Robot is at x=-8.521 y=-3.903 h=2.307.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 540.  Robot status is :STOPPED.
Time is 69153.92
```

```
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 541.  Robot status is :STOPPED.
Time is 69157.50
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.4235987755982901.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 542.  Robot status is :STOPPED.
Time is 69160.97
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 543.  Robot status is :STOPPED.
Time is 69168.18
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 544.  Robot status is :STOPPED.
Time is 69172.10
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 545.  Robot status is :STOPPED.
Time is 69175.80
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 546.  Robot status is :STOPPED.
Time is 69194.32
Robot is at x=-8.755 y=-3.604 h=2.165.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.8471975511965801.
```

```
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 547.  Robot status is :MOVING.
Time is 69201.53
Robot is at x=-8.919 y=-3.306 h=2.028.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 548.  Robot status is :MOVING.
Time is 69214.17
Robot is at x=-9.262 y=-1.940 h=1.639.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 549.  Robot status is :STOPPED.
Time is 69219.52
Robot is at x=-9.250 y=-1.561 h=1.446.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -6.567051725173267E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 550.  Robot status is :MOVING.
Time is 69230.35
Robot is at x=-9.220 y=-0.522 h=1.440.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.4245736962877518 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 551.  Robot status is :MOVING.
Time is 69239.18
Robot is at x=-9.182 y=0.215 h=1.622.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.4456215638571772 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.4245736962877518.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 552.  Robot status is :MOVING.
Time is 69258.67
Robot is at x=-9.258 y=0.910 h=1.758.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.387039115611643 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 553.  Robot status is :MOVING.
Time is 69271.15
```

```
Robot is at x=-9.699 y=2.234 h=2.067.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.4737320167650525 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 554.  Robot status is :MOVING.
Time is 69279.88
Robot is at x=-10.012 y=2.902 h=1.923.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 555.  Robot status is :MOVING.
Time is 69288.45
Robot is at x=-10.198 y=3.574 h=1.724.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.294739114875735 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 556.  Robot status is :MOVING.
Time is 69296.75
Robot is at x=-10.308 y=4.224 h=1.798.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.2626003893884954 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.294739114875735.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 557.  Robot status is :MOVING.
Time is 69306.45
Robot is at x=-10.479 y=4.861 h=1.881.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1676934643079202 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 558.  Robot status is :STOPPED.
Time is 69329.08
Robot is at x=-11.391 y=6.990 h=2.030.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 559.  Robot status is :STOPPED.
Time is 69332.43
Robot is at x=-11.391 y=6.990 h=2.030.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
```

```
Grabbing a frame.  Pan angle is 0.8471975511965801.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 560.  Robot status is :STOPPED.
Time is 69335.55
Robot is at x=-11.391 y=6.990 h=2.030.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 561.  Robot status is :STOPPED.
Time is 69342.37
Robot is at x=-11.391 y=6.990 h=2.030.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.4464055147217274 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 562.  Robot status is :MOVING.
Time is 69348.97
Robot is at x=-11.574 y=7.277 h=2.161.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.2895107439206765 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 563.  Robot status is :MOVING.
Time is 69357.22
Robot is at x=-11.954 y=7.791 h=2.254.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.2068011805833163 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.2895107439206765.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 564.  Robot status is :MOVING.
Time is 69363.78
Robot is at x=-12.321 y=8.214 h=2.310.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1478348468931663 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 565.  Robot status is :MOVING.
Time is 69375.65
Robot is at x=-13.246 y=9.126 h=2.382.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -9.624018145127389E-2 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.
```

```
Beginning cycle 566.  Robot status is :STOPPED.
Time is 69394.70
Robot is at x=-15.121 y=10.456 h=2.664.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Aiming camera.
Planning.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 567.  Robot status is :STOPPED.
Time is 69398.12
Robot is at x=-15.121 y=10.456 h=2.664.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Grabbing a frame.  Pan angle is -1.27079632679487.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 568.  Robot status is :STOPPED.
Time is 69401.33
Robot is at x=-15.121 y=10.456 h=2.664.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 569.  Robot status is :STOPPED.
Time is 69408.30
Robot is at x=-15.121 y=10.456 h=2.664.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.625 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 570.  Robot status is :MOVING.
Time is 69414.83
Robot is at x=-15.384 y=10.671 h=2.423.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.5707832784614242 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 571.  Robot status is :MOVING.
Time is 69423.08
Robot is at x=-15.831 y=11.128 h=2.252.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 0.1319691111338903 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 572.  Robot status is :MOVING.
Time is 69431.20
Robot is at x=-16.221 y=11.635 h=2.209.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 3.311880851168603E-2 10.0))
```

```
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is -0.1319691111338903.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 573.  Robot status is :MOVING.
Time is 69439.08
Robot is at x=-16.551 y=12.087 h=2.200.
Current plan: (GOTO ROBOT HOME-BASE)
*** Waiting for vision data ***
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 574.  Robot status is :STOPPED.
Time is 69458.38
Robot is at x=-16.691 y=12.282 h=2.186.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.238278494099184 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 575.  Robot status is :MOVING.
Time is 69464.78
Robot is at x=-16.911 y=12.542 h=2.279.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1357739896603243 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 576.  Robot status is :MOVING.
Time is 69472.90
Robot is at x=-17.324 y=13.004 h=2.322.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -9.360594910351061E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.1357739896603243.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 577.  Robot status is :MOVING.
Time is 69480.65
Robot is at x=-17.712 y=13.408 h=2.348.
Telemetry: :BOULDER-DATA
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -6.830793808032753E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 578.  Robot status is :MOVING.
Time is 69508.68
Robot is at x=-19.681 y=15.282 h=2.308.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -0.1367909831857248 10.0))
Grabbing a frame.
```

```
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 0.1357739896603243.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 579.  Robot status is :MOVING.
Time is 69527.40
Robot is at x=-20.062 y=15.664 h=2.367.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -7.027590414718254E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 580.  Robot status is :MOVING.
Time is 69539.05
Robot is at x=-20.979 y=16.528 h=2.441.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 1.560791744181067E-2 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 581.  Robot status is :MOVING.
Time is 69547.12
Robot is at x=-21.415 y=16.940 h=2.371.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -9.395634207015569E-2 10.0))
Aiming camera.
Planning.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 582.  Robot status is :MOVING.
Time is 69555.08
Robot is at x=-21.851 y=17.352 h=2.401.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -6.443244872327947E-2 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 9.395634207015569E-2.
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 583.  Robot status is :MOVING.
Time is 69562.82
Robot is at x=-22.252 y=17.713 h=2.416.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -5.344640832333702E-2 10.0))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 584.  Robot status is :MOVING.
Time is 69574.47
Robot is at x=-23.194 y=18.519 h=2.453.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((2.0 -1.753698845261997E-3 10.0))
Grabbing a frame.
Warning: grabbed a vision frame while robot was in motion.
Pan angle is 9.395634207015569E-2.
```

```
Stereo processing vision frame buffer 0 ... done.
Cycle complete.  Robot status is: :MOVING.

Beginning cycle 585.  Robot status is :MOVING.
Time is 69582.18
Robot is at x=-23.631 y=18.869 h=2.483.
Current plan: (GOTO ROBOT HOME-BASE)
Moving robot: ((1.775507495164106 3.181446071969418E-2
8.877537475820528))
Boulder processing a vision frame ... done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 586.  Robot status is :STOPPED.
Time is 69604.13
Robot is at x=-25.253 y=20.138 h=2.471.
Current plan: (GOTO ROBOT HOME-BASE)
*** Arrived at HOME-BASE. ***
Aiming camera.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 587.  Robot status is :STOPPED.
Time is 69640.33
Robot is at x=-25.253 y=20.138 h=2.471.
Current plan: (DELIVER-SAMPLE RED 3)
*** Delivering 3 RED rocks. ***
New sample status:
Storage available: 10
Samples available: ((BLUE 15) (GREEN 18) (RED 17))
Samples collected: ((BLUE 0) (GREEN 0) (RED 0))
Grabbing a frame.  Pan angle is -3.181446071969418E-2.
All tasks done.
Cycle complete.  Robot status is: :STOPPED.

Beginning cycle 588.  Robot status is :STOPPED.
Time is 69642.10
Robot is at x=-25.253 y=20.138 h=2.471.
Current plan: LISP:NIL
No more tasks.
All tasks done.

*** Sequence halted. ***

Simulator stats:
Elapsed time: 5153.500
Time in motion: 1393.200
Simulator run time: 2031.950
GC time: 740.667
Telemetry time: 516.767
Cumulative distance: 277.259 m
Average speed while moving: 0.199 m/s
Number of segments: 368
Number of stops: 45
Number of timeouts: 36
LISP:NIL
?
```

The final configuration is shown in figure 5-13.

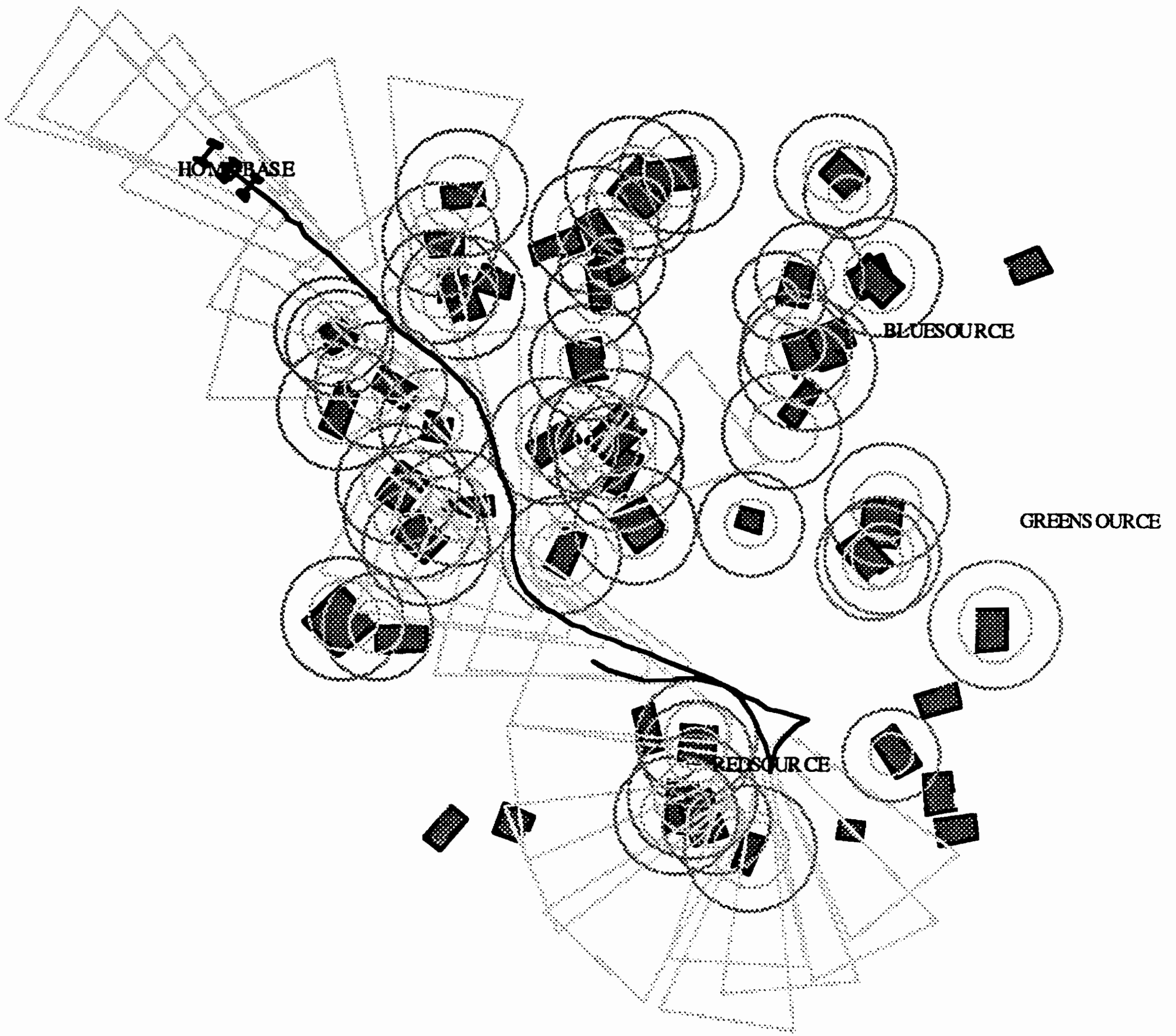


Figure A-13: The robot completes its final task.

Vita

Erann Gat received a B.S.E.E. Summa Cum Laude in 1985, an M.S. in Computer Science in 1987, and a Ph.D. in Computer Science in 1991, all from the Virginia Polytechnic Institute and State University. From 1979 to 1982 he worked for the American Museum of Science and Energy programming microcomputers for energy-related exhibits. In 1983 and 1984 he worked for the IBM corporation doing systems programming and bipolar VLSI integrated circuit design. Between 1985 and 1988 he worked as an employee of and later as a consultant for the Inland Motor Corporation of Radford, Virginia, doing research in fiber optic sensors. This work resulted in two United States Patents, one for an error detection system for a binary fiber optic sensor, and one for a wavelength-division multiplexing system using interference filters. Dr. Gat has also consulted for BEI Motion Systems of Carlsbad, California, and Hughes Research Laboratories in Malibu. Dr. Gat is currently a member of the technical staff at the California Institute of Technology Jet Propulsion Laboratory (JPL) where he has been working on autonomous robots since 1988.