

The Pennsylvania State University
The Graduate School

A COGNITIVE ROBOTIC SYSTEM BASED ON THE SOAR
COGNITIVE ARCHITECTURE FOR MOBILE ROBOT
NAVIGATION, SEARCH, AND MAPPING MISSIONS

A Dissertation in
Aerospace Engineering
by
Scott D. Hanford

© 2011 Scott D. Hanford

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2011

The dissertation of Scott D. Hanford was reviewed and approved* by the following:

Lyle N. Long
Distinguished Professor of Aerospace Engineering and Mathematics
Dissertation Advisor, Chair of Committee

Victoria A. Braithwaite
Professor of Fisheries and Biology

Richard A. Carlson
Professor and Associate Head of Psychology

Robert G. Melton
Professor of Aerospace Engineering

George A. Lesieutre
Professor and Head of Aerospace Engineering

*Signatures are on file in the Graduate School.

Abstract

Most unmanned vehicles used for civilian and military applications are remotely operated or are designed for specific applications. As these vehicles are used to perform more difficult missions or a larger number of missions in remote environments, there will be a great need for these vehicles to behave intelligently and autonomously. Cognitive architectures, computer programs that define mechanisms that are important for modeling and generating domain-independent intelligent behavior, have the potential for generating intelligent and autonomous behavior in unmanned vehicles. The research described in this presentation explored the use of the Soar cognitive architecture for cognitive robotics.

The Cognitive Robotic System (CRS) has been developed to integrate software systems for motor control and sensor processing with Soar for unmanned vehicle control. The CRS has been tested using two mobile robot missions: outdoor navigation and search in an indoor environment. The use of the CRS for the outdoor navigation mission demonstrated that a Soar agent could autonomously navigate to a specified location while avoiding obstacles, including cul-de-sacs, with only a minimal amount of knowledge about the environment. While most systems use information from maps or long-range perceptual capabilities to avoid cul-de-sacs, a Soar agent in the CRS was able to recognize when a simple approach to avoiding obstacles was unsuccessful and switch to a different strategy for avoiding complex obstacles.

During the indoor search mission, the CRS autonomously and intelligently searches a building for an object of interest and common intersection types. While searching the building, the Soar agent builds a topological map of the environment using information about the intersections the CRS detects. The agent uses this topological model (along with Soar's reasoning, planning, and learning mechanisms) to make intelligent decisions about how to effectively search the building. Once the object of interest has been detected, the Soar agent uses the topological

map to make decisions about how to efficiently return to the location where the mission began. Additionally, the CRS can send an email containing step-by-step directions using the intersections in the environment as landmarks that describe a direct path from the mission's start location to the object of interest.

The CRS has displayed several characteristics of intelligent behavior, including reasoning, planning, learning, and communication of learned knowledge, while autonomously performing two missions. The CRS has also demonstrated how Soar can be integrated with common robotic motor and perceptual systems that complement the strengths of Soar for unmanned vehicles and is one of the few systems that use perceptual systems such as occupancy grid, computer vision, and fuzzy logic algorithms with cognitive architectures for robotics. The use of these perceptual systems to generate symbolic information about the environment during the indoor search mission allowed the CRS to use Soar's planning and learning mechanisms, which have rarely been used by agents to control mobile robots in real environments. Additionally, the system developed for the indoor search mission represents the first known use of a topological map with a cognitive architecture on a mobile robot. The ability to learn both a topological map and production rules allowed the Soar agent used during the indoor search mission to make intelligent decisions and behave more efficiently as it learned about its environment. While the CRS has been applied to two different missions, it has been developed with the intention that it be extended in the future so it can be used as a general system for mobile robot control. The CRS can be expanded through the addition of new sensors and sensor processing algorithms, development of Soar agents with more production rules, and the use of new architectural mechanisms in Soar.

Table of Contents

List of Figures	ix
List of Tables	xv
Acknowledgments	xvi
Chapter 1	
Introduction	1
Chapter 2	
Soar and unmanned vehicles	4
2.1 Usefulness of computational psychology	
tools for unmanned vehicles	4
2.2 Common cognitive architectures	7
2.3 Cognitive architectures for robotics	10
2.4 The Soar architecture	13
2.4.1 Soar details	13
2.4.2 Soar for unmanned vehicles	21
2.5 Test missions	24
2.5.1 Navigation while avoiding obstacles	24
2.5.2 Search in an unknown indoor environment	25
Chapter 3	
Hardware	27
3.1 SuperDroid	27
3.2 Processors	27
3.2.1 Laptop	28
3.2.2 BrainStem module	28

3.2.3	MD22 motor driver	30
3.3	Sensors	30
3.3.1	Sonar	31
3.3.2	Infrared	31
3.3.3	Compass	32
3.3.4	GPS	32
3.3.5	Wheel encoders	33
3.3.6	Camera	34

Chapter 4

	Algorithms	37
4.1	State estimation	37
4.2	Motor control	39
4.3	Maps for mobile robots	40
4.3.1	Background	40
4.3.1.1	Metric	41
4.3.1.2	Topological	42
4.3.1.3	Hybrid	44
4.3.2	Maps in the CRS	48
4.3.2.1	Occupancy Grid Algorithm	48
4.3.2.2	Inverse Sensor Models	50
4.3.3	Implementation and Results	52
4.4	Hough transform	56
4.4.1	Background	56
4.4.2	Implementation and results	56
4.5	Intersection Detection	59
4.5.1	Generating semantic spatial information	59
4.5.2	Implementation	60
4.5.2.1	Virtual rangefinder	60
4.5.2.2	Fuzzy logic system	60
4.5.3	Results	63
4.6	SIFT	63
4.6.1	SIFT Background	64
4.6.2	Applications of SIFT	65
4.6.3	SIFT in the CRS	66

Chapter 5

	Outdoor navigation	69
5.1	CRS description for outdoor navigation	69
5.2	Description of Soar agents	71

5.2.1	Original Soar agent	71
5.2.2	Updated Soar agent	73
5.3	Tests and results	75
5.3.1	Results for original Soar agent	76
5.3.2	Results for updated Soar agent	77
5.4	Discussion	79

Chapter 6

	Indoor search	82
6.1	Hardware for indoor search mission	82
6.2	Software and algorithms	84
6.3	Description of Soar agent	86
6.3.1	Top-level operators	87
6.3.2	Input to and output from agent	88
6.3.3	Working memory	90
6.3.4	Search	93
6.3.4.1	Navigate	93
6.3.4.2	Setting plans in intersections	99
6.3.4.3	Performing a navigation plan	103
6.3.5	Return home	108
6.3.5.1	Decide initial return action	108
6.3.5.2	The navigate and perform plan operators	109
6.3.5.3	Setting plans in intersections	110
6.3.5.4	Ending the test	112
6.4	Results	113
6.4.1	Integration of Soar with the CRS	113
6.4.2	Hallway tests	132
6.4.2.1	First test	133
6.4.2.2	Second test	140
6.4.3	Test with loop	145
6.4.4	Simulation in more complicated environment	150
6.5	Discussion	166

Chapter 7

	Conclusion	172
7.1	Summary	172
7.2	Conclusions	173
7.3	Future work	176

Appendix A	
Example BrainStem TEA program	179
Appendix B	
Operator hierarchy for indoor search agent	183
Appendix C	
Explanation of selected Soar productions	194
C.1 State elaboration	194
C.2 Operator proposal	196
C.3 Operator preference	197
C.4 Operator application	199
C.5 Chunk	201
Appendix D	
Trace of planning	203
Bibliography	206

List of Figures

2.1	An example graph of working memory showing that a red block is on a black table. Labels in circles (S1, B1, and T1) are identifiers. Words preceded by a '^' character (object, on, type, and color) are attributes. Red, block, black, and table are values that are constant. T1 is the value associated with the ^on attribute, but is an identifier instead of a constant.	15
2.2	The phases of the Soar decision cycle	17
2.3	Overview of the traditional Soar architecture	19
2.4	Overview of the updated Soar architecture	20
3.1	A picture of the SuperDroid robot, shown in an office doorway. A 12-inch (about 30 cm) silver ruler is on ground in front of the SuperDroid.	28
3.2	A picture of the BrainStem module is shown.	29
3.3	A picture of the MD22 motor driver.	30
3.4	A picture of the SRF05 sonar distance sensor.	31
3.5	Two types of infrared distance sensors	32
3.6	A picture of the CMPS03 electronic compass.	33
3.7	A picture of the GPS unit.	34
3.8	A picture of the encoder kit with the circuit board holding two photoreflectors shown at the top and the encoder disk shown in the middle.	35
3.9	A picture of the webcam.	36
4.1	Common types of intersections	48
4.2	A comparison of sonar (left) and IR (right) perceptual fields.	52
4.3	Occupancy grid generated by a stationary robot.	53
4.4	Occupancy grid maps of a hallway with an open office door in the right wall	54

4.5	Occupancy grid maps of different intersection types. The green arrows represent the position and orientation of the robot on each occupancy grid.	55
4.6	Hough transform results	58
4.7	The 12 groups for which the average rangefinder distances are calculated are shown. The rangefinder is facing the top of the figure and ‘L,’ ‘F,’ and ‘R’ represent that left, front, and right groups. . .	61
4.8	An example membership function to calculate the certainty that there is a wall to the left of the robot.	62
4.9	SIFT matches between a training image (left) and test image (right) used to recognize a book.	68
4.10	SIFT matches between a training image (left) and test image (right) used to recognize a doll.	68
5.1	A schematic of the hardware used onboard the SuperDroid and their connections. Inputs to the BrainStem module and laptop are shown as red lines. Output from the laptop and Devantech Motor Driver are shown by the green lines. The black lines represent power supplied to the BrainStem module and the Devantech Motor Driver.	70
5.2	Integration of the Soar architecture and other software in the CRS.	71
5.3	The hierarchical structure of the avoid and goToTarget abstract operators and their low-level, or primitive, operators of the original Soar agent is shown. The abstract operators are shown in the boxes with solid borders and the low-level operators are shown in boxes with dashed borders.	72
5.4	The hierarchical structure of the operators and their low-level operators is shown. The abstract operators are shown in the boxes with solid borders and the low-level operators are shown in boxes with dashed borders.	74
5.5	A picture of the top level of the parking garage on which the tests described in this paper were performed. This picture was taken from a location in the lower left corner of Figures 5.6, 5.7, and 5.8. .	76
5.6	The results for a test with simple obstacles are shown. The hexapod started in the lower left corner of the figure at the green circle and was able to successfully avoid the obstacles and reach the target location by switching between the avoid and goToTarget problem spaces.	77

5.7	The results of a test with a cul-de-sac obstacle are shown. The SuperDroid started in the upper left corner of the figure at the green circle and was not able to successfully reach the target location because it became stuck in the cul-de-sac in the upper left corner of the figure.	78
5.8	The results of the test of the SuperDroid, in which the updated Soar agent navigates to the target location in the presence of a cul-de-sac, are shown.	78
6.1	A schematic of the hardware used onboard the SuperDroid and their connections. Inputs to the BrainStem modules and laptop are shown as red lines. Outputs from the laptop, BrainStem modules, and Devantech Motor Driver are shown as green lines. The black lines represent power supplied to the BrainStem modules and the Devantech Motor Driver. The second BrainStem GP 2.0 module communicates with and receives power from the first BrainStem GP 2.0 module using its I2C ports. There is a USB to serial converter between the laptop and first BrainStem module.	83
6.2	The top level operators for the indoor search agent are shown. Abstract operators are shown in the boxes with solid borders and the primitive operators are shown in boxes with dashed borders.	87
6.3	The three abstract operators for the search operator of the indoor search agent are shown.	93
6.4	The operators used in the followMiddle abstract operator are shown.	95
6.5	The operators used to generate a navigation plan are shown.	100
6.6	The operators used to perform a navigation plan are shown.	103
6.7	The abstract operators for the returnHome operator.	108
6.8	CRS and use of the Soar decision cycle	115
6.9	CRS and use of the Soar decision cycle	116
6.10	Figure 1 of 3 showing the interaction between the CRS and the Soar agent (continued in Figure 6.11).	117
6.11	Figure 2 of 3 showing the interaction between the CRS and the Soar agent (continued in Figure 6.12).	119
6.12	Figure 3 of 3 showing the interaction between the CRS and the Soar agent.	120
6.13	Figure 1 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.14).	121
6.14	Figure 2 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.15).	123

6.15	Figure 3 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.16).	124
6.16	Figure 4 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.17).	125
6.17	Figure 5 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.18).	126
6.18	Figure 6 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission.	127
6.19	Figure 1 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.20).	128
6.20	Figure 2 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.21).	129
6.21	Figure 3 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.22).	130
6.22	Figure 4 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.23).	132
6.23	Figure 5 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.24).	133
6.24	Figure 6 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator. . . .	134
6.25	A map of the hallways used and arrows, labeled with numbers, indicating the order of movements the robot made through the environment during the test described in Section 6.4.2.1. The robot started at the green triangle and found the object of interest at the black 'X.'	135
6.26	A depiction of the topological map generated by the agent during the test described in Section 6.4.2.1. The bold numbers 1–5 are the identification numbers given to each node in the map. A connection from one node to a second node is represented by an arrow pointing to the second node. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'	136
6.27	The SIFT matches between the training image and an image used to identify the doll during the test are shown.	138

6.28	A map of the hallways used and arrows, labeled with numbers, indicating the order of movements the robot made through the environment during the test described in Section 6.4.2.2. The robot started at the green triangle and found the object of interest at the black 'X.'	141
6.29	A depiction of the topological map generated by the agent during the test described in Section 6.4.2.2. The bold numbers 1–5 are the identification numbers given to each node in the map. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'	142
6.30	A picture of the robot in the environment used for the test described in Section 6.4.3.	145
6.31	A map of the environment used for the test described in Section 6.4.3. The numbered arrows represent the order of the robot's movements during the test.	147
6.32	A depiction of the topological map generated by the agent during the test described in Section 6.4.3. The bold numbers 1–8 are the identification numbers given to each node in the map. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'	148
6.33	The topological map used for the simulations described in Section 6.4.4. The part of the map represented by a dashed line was not explored by the simulated robot.	151
6.34	Figure 1 of 6 showing planning in simulated test (continued in Figure 6.35).	153
6.35	Figure 2 of 6 showing planning in simulated test (continued in Figure 6.36).	156
6.36	Figure 3 of 6 showing planning in simulated test (continued in Figure 6.37).	158
6.37	Figure 4 of 6 showing planning in simulated test (continued in Figure 6.38).	160
6.38	Figure 5 of 6 showing planning in simulated test (continued in Figure 6.39).	161
6.39	Figure 6 of 6 showing planning in simulated test.	163
6.40	Figure 1 of 2 for the simulation performed with knowledge of chunks generated in the previous simulation described in Figures 6.34–6.39 (continued in Figure 6.41).	165

6.41 Figure 2 of 2 for the simulation performed with knowledge of chunks
generated in the previous simulation described in Figures 6.34–6.39. 165

List of Tables

4.1	Confidence values for the presence of common intersection types, detected from occupancy grids	64
6.1	Sensors and uses for sensor information in indoor search mission . .	84
6.2	Algorithms and uses of results for indoor search mission	85

Acknowledgments

I have been lucky to have the support of many people while pursuing this degree.

I would like to thank Dr. Long for being my advisor and for all of his guidance throughout my time at Penn State. I am especially appreciative of being able to maintain a balance between working on my degree and still being able to enjoy time with my family.

I'd also like to thank my committee members, Dr. Victoria Braithwaite, Dr. Richard Carlson, and Dr. Robert Melton, for their comments and enthusiasm.

I'd like to acknowledge the Soar group at the University of Michigan for making the Soar architecture available and providing documentation for new users.

Thanks to my fellow graduate students, Oranuj Janrathitikarn and Greg Sinsley, who have helped with various aspects of my research: Oranuj with some software for integrating hardware and software components and Greg with the fuzzy logic system.

I would like to acknowledge several funding sources that have provided support through my graduate studies: the National Defense Science and Engineering Graduate Fellowship program, the National Science Foundation Graduate Research Fellowship program, the Aerospace Engineering Department, the Signal Processing Department at Penn State's Applied Research Laboratory, and finally the Applied Research Laboratory's Exploratory and Foundational Research Program.

I would also like to acknowledge the help and support of many family and friends. I wish I could better express how grateful I am to have them in my life. Additionally, several friends and family members that have completed their own Ph.D. programs have provided me inspiration and hope that I could finish my own program.

I can't imagine there being more supportive and loving parents than my parents, Keith and Pat Hanford. Thank you for everything you have done for me. I'd also like to thank my Mom for coming to help us watch Carter for a good part of her summer.

I'd like to thank my sister and her husband, Kim and Chuck Scibetta, for

everything they've done to help us get through my grad school career.

Thanks to the extended Danforth family for welcoming me into their family and all of their support of Mandy, Noah, Carter, and I.

Mandy, thanks for being an incredible wife and mother. Thanks for believing in me when I got discouraged and really, just for everything. This has been a long process and I'm glad you were there for it- I certainly couldn't have done it without you.

To Noah and Carter, thanks for the reminders about the things that matter the most. There is nothing quite like your smiles for cheering me up.

Introduction

Mobile robots and unmanned vehicles have continued to become more prevalent for a variety of civilian and military applications. Many of these vehicles, however, currently require significant human supervision or are teleoperated. There are several challenges associated with controlling unmanned vehicles by teleoperation. The delays and limited bandwidth typical of communication with robots such as under-water vehicles and mobile robots used for planetary exploration can make teleoperation of these vehicles challenging. Teleoperation can require significant effort and attention on the part of the operator, resulting in a high operator to robot ratio (even requiring multiple operators to control one robot). It can be difficult for an operator to understand the robot's situation well enough to control a robot through teleoperation [1].

Increasing a robot's autonomy, which is described as the ability to operate "in the real-world environment without any form of external control for extended periods of time [2]," can overcome some of the difficulties associated with teleoperation. Even small increases in autonomous capabilities could be beneficial by allowing a robot's operator to focus on high-level decisions, interact with more than one robot without the risk of cognitive overload [3], or to transition from a relationship that requires an operator to control or supervise a robot to a collaborative relationship. Highly publicized examples of successful unmanned vehicles with autonomous capabilities include vehicles from the 2005 DARPA Grand Challenge and the 2007 DARPA Urban Grand Challenge and NASA's Mars Exploration Rovers. Although vehicles that participated in these DARPA competitions demonstrated impressive

behavior [4], [5], there were still collisions between vehicles in the controlled environment of the Urban Grand Challenge [6].

While autonomy can be a useful characteristic for unmanned vehicles, there are other important capabilities to consider. For example, it is possible for a system (such as a simple animal) to be completely autonomous, but not generate intelligent behavior. Another DARPA program, the Learning Applied to Ground Vehicles program (LAGR), has encouraged the development of certain aspects of intelligence, particularly learning, for unmanned vehicles [7]. Intelligence can be thought of as “a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience [8].” The LAGR program participants developed software that was tested on a common robotic system. Each participating team’s software was tested on the time the robotic system took to complete an off-road outdoor navigation mission by reaching a specified goal location. The test environments featured complex obstacles such as cul-de-sacs and obstacles (such as vegetation, walls, and fences) that made it unreliable to use obstacle detection schemes tuned for specific kinds of obstacles. Each team’s software was used to perform the mission several times, allowing several types of learning to occur. Teams were able to develop models to predict the traversability of an area based on its appearance in an image, learn maps of the environment, and use these maps and the paths taken to the goal location in earlier tests to determine a better path to take to the goal location during the current test [9], [10].

Interest in robotic systems able to move beyond achieving useful behavior for specific applications or in specific environments to systems capable of generalized intelligence [11] has encouraged interest in robots capable of cognitive functions commonly attributed to humans. Useful cognitive capabilities commonly mentioned include learning about the environment [12], [9], [10], being able to use a large knowledge base that describes how to act in an environment based on the current context or situation [13], [12], [5], and being able to generate reactive, deliberate, and reflective behavior [14], [12]. The importance of a system being able to collaborate with human and robot teammates is also commonly mentioned [13], [1]. This collaboration can include learning from both robot and human teammates [13], [12], anticipating and understanding the actions of others [6], [5],

[14], being able to share knowledge about the environment [12], and explaining its current behavior and the reasons it chose to perform that behavior [12]. Robotic systems, such as those in the Urban Grand Challenge, where vehicles had to generate complex, situation dependent behavior to follow common driving rules, and the LAGR program, which required systems to learn from previous experiences in their environment, have been able to demonstrate some of these capabilities. Other systems that have been developed with the goal of creating robots capable of cognitive capabilities include CARACaS [14], CoSy [12], and 4D/RCS [9], [15].

In addition to robotic systems that aim to generate intelligent behavior, there is also research in the field of computational psychology that has resulted in the creation of computer algorithms, such as cognitive architectures, that are capable of modeling intelligent behavior. Since these computational psychology models can be used to predict behavior in intelligent entities, there has been interest in using these algorithms to intelligently control unmanned vehicles. This dissertation will describe the development of a system called the Cognitive Robotic System, or CRS [16], [17], to use the Soar cognitive architecture [18] for control of unmanned vehicles.

Chapter 2 describes the characteristics of cognitive architectures, and particularly Soar, that make these architectures interesting candidates for control of unmanned vehicles. The motivation for two robotic missions (outdoor navigation and search in a building) used to test the CRS is also discussed. Chapter 3 describes the robotic hardware used in the CRS and Chapter 4 describes the software algorithms used for perception and motor control in the CRS. Chapters 5 and 6 discuss the implementations of the CRS used for the two robotic missions. The integration of the hardware and software components, the development of the Soar agents, the details of how the CRS uses the Soar architecture, and the results of the tests are all described. Chapter 7 summarizes the contributions of this research and discusses potential future development of the CRS.

Soar and unmanned vehicles

2.1 Usefulness of computational psychology tools for unmanned vehicles

Computational psychology tools (also called computational cognitive models) describe cognition using computer algorithms and programs. Since these models are executed on computers and can predict or generate behavior, they hold potential for generating cognitive behavior for the control of unmanned vehicles.

While there are several types of computational cognitive models [19], one way to classify these systems is as either symbolic or subsymbolic. A useful way to explain the distinction between symbolic and subsymbolic systems is the relationship between the level of semantic interpretation (the level at which meaningful concepts are described) and the formal representation for the system's processing [20]. For symbolic systems, the level of semantic interpretation and the formal representation for processing are both at the level of symbols: the symbols are the meaningful concepts and the processing involves manipulation of symbols. It has been argued that the processing of symbols is vital for complex cognition [21]. Laird stated that "the generality and power of symbolic representations and processing are unmatched and our ability to compose symbolic structures is a hallmark of human-level intelligence [22]." For subsymbolic systems, the formal representation for processing occurs below the level at which the concepts of interest are described. For example, the level of semantic interpretation for a digital image

could be objects in the image (e.g., doors or trashcans). A symbolic processing system would manipulate symbols that represent these objects in the image while a subsymbolic processing system would manipulate the pixels that make up these objects.

The most common subsymbolic systems are connectionist models such as neural networks and the most common symbolic systems are cognitive architectures. Kelley and Long [11] provided a nice summary of the strengths and weaknesses of neural networks: given proper network topology and training, these subsymbolic systems can perform perception and classification of information in the presence of noise. Although it can be difficult for traditional neural networks to achieve this performance for sets of data different than the training examples, there are newer types of neural networks that show promise for more successful performance in this situation.

While neural networks have most commonly been used for perception tasks, cognitive architectures have proven to be useful for modeling intelligent behavior in diverse domains. In part because of this ability to generate intelligent behavior, cognitive architectures are an interesting candidate for control of unmanned vehicles [13], [1], [23], [24], [25], [26], [27], [16]. The importance of using both symbolic and subsymbolic features for the intelligent control of mobile robots has been described by Kelley and Long [11] and both types of features are used in the Cognitive Robotic System described in this dissertation.

Newell proposed the development of cognitive architectures that would define the fixed processes that their designers believed to be important for intelligent behavior [21]. These architectures provide mechanisms important to cognition, regardless of the situation, task, or problem to which the architecture is applied. The developer of a model or agent of the architecture must encode knowledge that will use the architectural mechanisms to perform the task of interest. In other words, an architecture needs content (knowledge) that can be processed using architectural mechanisms to produce behavior [21], [28].

Systems capable of general intelligence will need to be able to generate intelligent behavior in different domains; these systems will need to be more than specialized algorithms developed for specific tasks [13]. Cognitive architectures that implement symbolic processing could take advantage of the generality and power

of symbolic processing [22] to control robots for problems in several domains. An unmanned system could use the same cognitive architecture and general knowledge for all tasks of interest and add new knowledge specific to additional tasks or missions through additional programming, instruction, or learning. In addition to generating intelligent behavior in multiple domains, cognitive architectures may be useful for developing systems with cognitive capabilities that are common for humans but can be difficult to realize in unmanned systems (such as learning, anticipating the actions of human teammates, or performing reflective reasoning about their own behaviors) [13].

There are also potential benefits to using robots that have models of how humans act or represent information in a way similar to humans. Researchers have argued that having robots that use processes similar to humans to generate behavior would facilitate effective human-robot interaction [1], [13]. A robot controlled by a cognitive architecture may act in a way that enables a human (or even a robotic) teammate to understand why the robot is behaving as it is and make it more likely the teammate could predict the robot’s behavior [13]. More effective human-robot interaction could lead to benefits such as increasing the number of robots a human operator could work with at once. Additionally, for some tasks it may be easier to create a model of how humans perform a task or to represent information in a way similar to how humans do that can be implemented on a robot than to develop a traditional robotic control algorithm to perform that same task [1]. For example, Trafton et al. [1] suggested that a robot searching for a hiding sniper might be most successful if it included a model with knowledge about how humans hide. Other algorithms may be easier to describe symbolically than numerically. Trafton et al. [1] also noted that “traditional AI spatial reasoning techniques do not adequately capture how people perform spatial reasoning; a model based on human spatial reasoning will provide some robust advantages over those systems that do not reason as a person would.” An application that illustrates the flexibility and usefulness of human spatial reasoning is the collaboration between astronauts and an observer on the earth. This collaboration can require mixing of frames of reference and taking the perspective of a person that may not be visible [1], [29].

2.2 Common cognitive architectures

The previous section described reasons that a cognitive architecture could be useful for control of an unmanned vehicle; this section describes some specific architectures that could be candidates for use with an unmanned vehicle. Since “no theory in the foreseeable future can hope to account for all of cognition [21],” many cognitive architectures have been developed. These different architectures often focus on modeling different aspects of cognition and model cognition at different levels of abstraction. The most well-known architectures are Soar [18], ACT-R [30], and EPIC [31]. Soar and ACT-R were originally developed with a focus on modeling central cognition such as problem solving and learning, while EPIC emphasized the importance of perceptual and motor systems to cognition.

Soar has roots in artificial intelligence [32] and the architectural mechanisms of Soar focus on functionality for problem solving and producing intelligent behavior. Soar uses a production system in which all of its knowledge about how to behave (production rules) and all of its knowledge about its current state are applied to demonstrate flexible and goal-driven behavior. A strength of the Soar architecture is modeling decision-making at a level of abstraction useful for performing high-level control of unmanned vehicles. Soar has historically been a symbolic architecture; however, the architecture has recently been extended to include non-symbolic mechanisms [22] useful for robot control [13]. (These new mechanisms will be discussed in more detail in Section 2.4.1.) Soar is best known for generating agents capable of intelligent behavior in simulations requiring human-level intelligence [33], [34].

ACT-R (Adaptive control of thought-rational) [30] has its roots in cognitive psychology and was initially inspired by a theory of human memory [35]. ACT-R models intelligent behavior using modules associated with regions of the brain and a central production system that has access to a subset of the information from the modules. Examples of these modules include an intentional module, which contains information about goals, a declarative memory module (facts), a visual module (for perception), and a manual motor module (for output from the architecture to an environment). Each module has a buffer that can store only one unit of declarative knowledge (called a chunk) at a time. Collectively, the information in the buffers

of all modules represents the state of the ACT-R model. The pattern matcher in the central production system searches for a production whose conditions are satisfied by the current state of the buffers. To support the symbolic processing that occurs in the production system, ACT-R uses subsymbolic processes to aid the retrieval of chunks from modules into buffers (memory activation) and the selection of a single production when the conditions of multiple productions are matched by the current state (using values that represent the expected utility of productions). ACT-R is well known for successfully modeling many aspects of human cognitive performance and for an algebra tutoring system [30].

The EPIC (Executive Process-Interactive Control) [31] architecture has emphasized that perception and action, in addition to central cognition, have an important role in cognition and task performance. EPIC has detailed mechanisms for sensory information processing (visual, auditory, and tactile) and motor (ocular, vocal, and manual) activity. EPIC models have successfully demonstrated multi-tasking behavior, which has been useful in the area of human computer interaction design [36] for designing applications such as telephone operator workstations and airplane cockpits.

Many other cognitive architectures including Polyscheme [37], a cognitive architecture which can use several specialists to integrate different types of reasoning, and Clarion [38], which uses both symbolic processing and neural networks, have also been developed. It is also possible to integrate components from different cognitive architectures. For example, the EASE [39] cognitive architecture integrated some of the strengths of several architectures: the peripheral processors (sensory, perceptual, and motor) of EPIC, the cognitive processor of Soar (including problem solving, planning, and symbolic learning), and a learning mechanism detailing human memory from ACT-R. As another example, the perception and motor modules in EPIC were used as inspiration for the corresponding modules in ACT-R, Clarion, and Soar [36].

Taatgen and Anderson [40] explained that it can take a large investment of time and effort to understand a cognitive architecture well enough to develop meaningful models using that architecture. This large investment makes it “nearly infeasible [40]” to compare multiple architectures. (It also makes access to a community of more advanced users and introductory and tutorial materials invaluable when

learning an architecture [41].) Therefore, making an informed decision about which cognitive architecture best suits a researcher’s needs can be quite challenging.

There have been several papers reviewing various cognitive architectures [41], [42], [43], [44]. The AMBR (Agent-Based Modeling and Behavior Representation) project [45] used a simulated air traffic control environment to perform a direct comparison of four architectures performing air traffic control tasks that involved multi-tasking and category learning. The models developed using each architecture were compared both to human data and to the models created using the other architectures. However, performing direct, detailed comparisons such as AMBR can be difficult, expensive, and time-consuming.

To help select an architecture, it may be useful to look at some of the constraints that the architecture applies on processing [46] or the applications for which the architecture is commonly used. For example, these constraints and applications help demonstrate some differences between Soar and ACT-R. Through its history, Soar’s focus has been on supporting the functional mechanisms needed to generate intelligent behavior. Alternately, the focus of ACT-R has been on the modeling of human behavior. Accordingly, Soar has been used to create models that perform “complex tasks, using and acquiring large store or knowledge across extended time spans, often requiring significant internal processing and planning [47].” ACT-R is commonly used to develop models able to predict aspects of human performance, such as reaction time, accuracy, and brain activity (using fMRI data), for specific tasks [40].

These different areas of focus may help explain some of the differences in constraints used by the Soar and ACT-R architectures, in particular the constraint on the access to information in working memory each architecture provides. Each ACT-R module has a buffer that can hold only one element of information. Since the buffer contents in ACT-R are used to determine productions that are relevant to the current situation, this constraint effectively limits the ACT-R model’s access to working memory. The desire to restrict the number of working memory element that can be considered at once is consistent with observations about human behavior [48], although ACT-R’s implementation may be more restrictive than those observations [46]. This limited access makes it difficult for ACT-R models to consider lots of information about the current situation. It has been

noted that this constraint “makes it difficult or impossible to perform human-level reasoning [49].” In contrast, the Soar architecture does not prescribe a limit on access to working memory (although some models have included more restrictive access [34]). This open access is consistent with the belief in Soar that all possible sources of knowledge should be used to reason about a problem, but can result in productions that have quite complex conditions that may not be representative of human performance [46]. While this may be a concern for modeling human performance with Soar, it can be beneficial when applying Soar to robotics.

2.3 Cognitive architectures for robotics

Researchers have used both popular cognitive architectures such as ACT-R [30] and Soar [18] and developed their own architectures to use on robots. There are challenges associated with using these architectures in real environments. For example, it can be particularly difficult to generate meaningful symbols that can be used for reasoning by the symbolic components of the cognitive architectures from (potentially noisy) sensor data or to perform some low-level tasks important for unmanned vehicles such as control of motors using cognitive architectures. To deal with some of the challenges of using cognitive architectures to control unmanned vehicles, some researchers have integrated additional systems with the cognitive architecture that complement the strengths of the cognitive architecture. The rest of this section describes systems that have used cognitive architectures to control unmanned vehicles.

Use of the Soar cognitive architecture to control physical robots was described as early as 1990 [50], [51]. Soar was used to control both a mobile robot that used sonar sensors and a robotic gripper to locate and pick up cups and a robotic manipulator arm with a vision system for a block manipulation task. While these projects illustrated the usefulness of Soar for cognitive robotics, they also showed the need for improved perceptual systems to fully take advantage of Soar’s capabilities [13].

After these two robotic systems, the Soar architecture was used to develop intelligent agents for the control of simulated unmanned vehicles. The agents have proven useful for generating realistic behavior in simulations requiring human-

level intelligence. TacAir-Soar and RWA-Soar agents were used to autonomously control fixed-wing and rotary-wing aircraft in simulated training environments [52], [33]. Soar agents (MOUTBots) were also used to generate behavior of simulated opponents in virtual immersive environments used by teams of U.S. marines for training for military operations on urbanized terrain [34]. Another project used Soar agents to control simulated air and ground vehicles [53]. The Soar agents in these projects demonstrated autonomous and intelligent behavior in a complex, albeit simulated, environments with real-time constraints. More recently, Soar has been used to control a mobile robot in both real and simulated indoor environments [13], [47] and on physical robots for research in human-robot collaboration for reconnaissance missions [3].

The Intelligent Systems Section at the Navy Center for Applied Research in Artificial Intelligence has used multiple cognitive architectures in its efforts to build robots that can think and act like humans with the goal of facilitating better and easier collaboration between humans and robots [1]. ACT-R, ACT-R/S (an extension of ACT-R that includes a theory of spatial reasoning) [1], and Polyscheme [37] have been used to perform research in the area of robots and spatial reasoning through the game of hide and seek and a perspective taking task. Additionally, Kennedy et al. [23] developed a robot capable of approaching or following a target (as in a reconnaissance mission) independently or collaboratively with humans using verbal and non-verbal communication. To accomplish this task, which is greatly dependent on the robot's ability in spatial reasoning, a three-layer architecture was developed. The bottom layer included sensors, effectors, and a non-cognitive system for standard robotic tasks (path planning, obstacle avoidance, and navigation), previously described by Schultz et al. [54]. The middle layer was a spatial support layer and included a cognitive map that represents space in a qualitative manner as a 2-D grid in which objects are placed using the metric information describing their locations. The 2-D cognitive map supports symbolic reasoning about relative locations of objects of interest (such as teammates, targets, and buildings). The top, or cognitive, layer included an ACT-R model.

More recently, this group has described the development of ACT-R/E (ACT-R/Embodied) [49], an architecture used for interaction between ACT-R and real-world robots and sensors. Some of ACT-R's existing modules have been modified

to allow for specific input from specific sensors and new modules have been integrated for interaction with the cognitive map described by Kennedy et al. [23] (spatial module) and the non-cognitive localization, navigation, and path planning system described by Schultz et al. [54] (pedal module). The ACT-R/E system has been combined with a system capable of tracking people using an omni-directional camera, a detection algorithm, and a particle filter and another system capable of estimating sound source positions using pairs of microphones and a sound localization algorithm [49]. An ACT-R/E model has been developed for conversation tracking: information about people and sound locations in the robot's environment is used by the agent to look at a person speaking in a dynamic environment with multiple people (who may be moving and interrupting other speakers) and other distractors. Another project used a ACT-R/E model of infant gaze-following to control a robot's eyes and head movements [55].

ADAPT is a robotic cognitive architecture that has been embedded in the Soar architecture by integrating Soar with the RS (robot schemas) language for sensorimotor activity, a theory of decomposition and reformulation, and a natural language system [27], [56]. The development of ADAPT has been motivated by the desire to address two inadequacies of cognitive architectures for robotics research: true concurrency and active perception. Concurrency allows the coordination of several behaviors (e.g., turning a camera to the left to keep an object in the camera's field of view while the robot holding the camera is turning to the right). Active perception can be considered perception in pursuit of a goal. Two examples of active perception useful for a robot would be the selective processing of only the important areas of an image and adjusting the parameters of a camera to focus on a particular object in the robot's environment. Recent research using the ADAPT system has studied the ability to understand perceived behaviors by other agents by attempting to generate the same behavior using operators of a Soar agent in a simulated world model [57]. Additionally, ADAPT has used Soar operators to activate robot schemas to generate robot behavior and implemented chunking, a learning mechanism in Soar, to build and maintain networks of robot schemas.

The Symbolic and Subsymbolic Robotic Intelligent Control System (SS-RICS) is a system that integrates several successful AI techniques (symbolic and subsym-

bolic processing, machine learning, and semantic networks) for control of physical mobile robots [24], [26], [58], [59]. SS-RICS has been built around a production system at the symbolic level and can also use AI techniques at both the subsymbolic (neural networks) and symbolic (machine learning) levels. Subsymbolic processing is used to process all sensor inputs. This processing can be used for tasks such as determining if measurements from a laser rangefinder can be considered as a group that may indicate an important item (e.g., a corner or a wall) in the robot’s environment [25]. Symbolic processing occurs in the form of a production system (similar to ACT-R) that has access to high-level symbolic memory that includes results from processing at the subsymbolic level.

2.4 The Soar architecture

While different cognitive architectures focus on different aspects of cognition, there are several strengths of Soar that make it a promising candidate for use in unmanned vehicles. This section describes the basics of the Soar architecture, focusing on the description of the traditional Soar architecture that uses symbolic processing for problem solving. The architectural mechanisms that help make Soar well suited for control of unmanned vehicles will then be described. While the research described in this dissertation used the traditional Soar architecture, the recent extensions to the Soar architecture will also be discussed.

2.4.1 Soar details

As described in Section 2.1, while the Soar architecture provides fixed mechanisms thought to be important for intelligent behavior, a Soar agent (or model) must also have knowledge that is able to use the mechanisms defined by the architecture. This section describes how the mechanisms of the Soar architecture use the knowledge in a Soar program to generate behavior by a Soar agent. The Soar Manual and Tutorial (available online with the Soar software [60]) and another introductory document [28] are useful sources to supplement the information presented below.

A Soar agent attempts to use the knowledge it has about its current situation,

or state, to apply operators to change its state in an effort to reach a goal. The traditional Soar architecture uses two types of memory, long-term knowledge in the form of production rules and short-term knowledge stored in working memory, to perform symbolic processing.

Working memory can be thought of as a graph structure that contains information about the current state of the Soar agent and its environment. There are parts of working memory called input and output links that can be use for communication of information between the Soar agent and an external environment. Each working memory element (WME) is an identifier-attribute-value triple. An identifier is a pointer to an object described in working memory, the attribute describes a characteristic of the object, and the value contains information about its corresponding attribute. A value can contain a constant or another identifier, allowing working memory to generate a hierarchical structure describing short-term knowledge. Figure 2.1 shows the working memory associated with a state S1: this example graph shows that a red block is on a black table. The individual working memory elements that describe this graph are:

- S1 ^object B1
- S1 ^object T1
- B1 ^color red
- B1 ^type block
- B1 ^on T1
- T1 ^color black
- T1 ^type table

In this list of working memory elements, the first member of each WME is the identifier, the second member of each WME is the attribute (the name of each attribute is preceded by the ^symbol), and the third member of each WME is the value.

A Soar agent is developed by writing productions. Each production has a set of conditions and a set of actions, similar to an if-then rule in a traditional

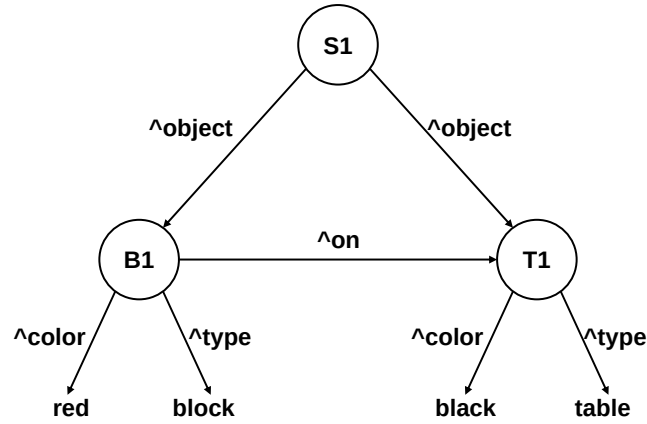


Figure 2.1: An example graph of working memory showing that a red block is on a black table. Labels in circles (S1, B1, and T1) are identifiers. Words preceded by a ‘^’ character (object, on, type, and color) are attributes. Red, block, black, and table are values that are constant. T1 is the value associated with the ^on attribute, but is an identifier instead of a constant.

programming language. The set of conditions of a production represents a situation in working memory for which the production is relevant and should be fired. All productions whose complete sets of conditions are matched are fired by executing the complete set of actions of that production. Productions can be used for several reasons: state elaboration (i.e. to add information or make inferences about the current state of working memory) or to propose, compare, or apply operators that make changes to working memory to move the agent closer to reaching its current goal.

The Soar architecture separates the proposal, selection, and application of operators. Each operator is implemented by productions that propose, compare, and apply operators. This separation allows more than one operator to be proposed at a time. All operators whose conditions are met by the current state are proposed with acceptable preferences so they are eligible for selection. Candidate operators (those with acceptable preferences) can then be evaluated by productions that can result in information about a single candidate operator (such as a best, reject, or worst preference) or a comparison between two candidate operators (candidate operator A is better or worse than candidate operator B for the current situation). Instead of restricting the architecture to the selection of a single operator based

on the operator's conditions, this separation allows the architecture to use more information than conditions of a production to select an operator. For example, the conditions for proposal of an operator can be general (to consider all relevant operators), while the comparison of candidate operators can be more specific to the agent's current situation. Once an operator has been selected, the production that applies that operator can be fired and make the changes in working memory prescribed by that production's set of actions.

The changes in working memory that result from the actions of production rules can be given two different types of support (i-support and o-support) that determine how long the changes persist. Changes that occur as a result of an action in a production that elaborates a state, proposes an operator, or generates a preference for a candidate operator receive i-support (instantiation-support). These i-supported changes persist only as long as the conditions of the production that generated the changes remain true. Once the working memory element(s) that satisfy a condition are no longer true, that production is retracted and the Soar architecture automatically undoes the changes made to working memory by that production's actions. However, changes in working memory that occur during the application of an operator are given o-support (operator-support). These changes persist until another operator application production explicitly undoes them (or the state that contains the working memory elements is removed from working memory). Appendix C provides examples of Soar productions that have actions with i-support as well as production that have o-supported actions.

The execution of a Soar agent is governed by a decision cycle in which the agent attempts to choose an operator to apply based on the information it knows about its current state. This cycle can be broken into five parts, as shown in Figure 2.2:

1. In the input phase, Soar can receive information from a simulated or real external environment (e.g. sensor input) that has been placed on its input-link in working memory.
2. In the second phase, the current state is elaborated and operators are proposed based on the contents of working memory. Productions that create preferences for candidate operators that have been proposed also fire. Productions fire (and retract) in parallel until no additional changes are possible

(a condition known as quiescence).

3. In this phase, one operator is selected if possible. If more than one candidate operator has been proposed in the second phase, Soar has a mechanism to attempt to choose the most appropriate operator for the current situation (based on operator preferences created in the second phase).
4. In the application phase, the selected operator makes changes to working memory that represent an attempt to move the agent closer to achieving its current goal. These changes can also result in additional production firings and retractions. Similar to the second phase, these firings and retractions continue until quiescence is reached. The changes specified by the operator application can encompass both direct changes (for internal problem solving) and indirect changes to the state (for agents in external environments, actions must be performed in external environments which can then be monitored for any resulting changes).
5. In the output phase, Soar can send commands to an external environment by using the agent's output-link (e.g. motor commands, verbal commands to other agents, or other actions).

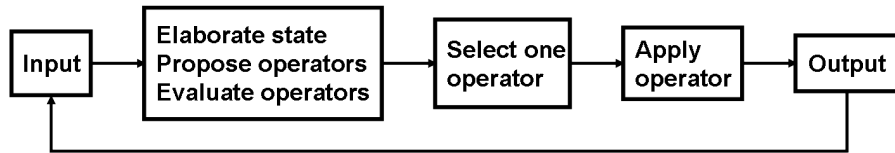


Figure 2.2: The phases of the Soar decision cycle

If a Soar agent does not have enough knowledge to select or apply an operator during its reasoning cycle, an impasse occurs in which Soar automatically creates a subgoal to resolve the impasse. There are several different impasse types: tie, conflict, constraint-failure, state no-change, and operator no-change. The tie and operator no-change impasses are used in the Soar agents described in this dissertation. An operator no-change impasse occurs when a new operator is selected but the agent is unable to apply the operator because no productions have all of their conditions met during the application phase. A tie impasse happens when the Soar

architecture is not able to select an operator from a set of more than one candidate operator because there is not enough information in preference memory to determine which candidate operator should be selected in the current situation. The automatic generation of a subgoal to resolve a tie impasse allows Soar agents to both reactively select an operator (when sufficient knowledge is available to choose one without the occurrence of a tie impasse) and deliberatively reason about which operator to select using planning (when the agent does not have enough knowledge to reactively select one operator and a tie impasse does occur).

When the Soar agent has an internal model of its environment so that it is able to reason about the effects of applying possible operators, simple look-ahead planning in which tied operators are simulated internally to determine the likely outcome of an operator before choosing to apply it in an external environment can be a useful approach for comparing candidate operators. When a Soar agent uses planning in an impasse, it is not creating a step-by-step plan to be followed blindly; it is using planning as a source of knowledge to choose an operator to apply in the current situation (an impasse has occurred because its other knowledge did not provide enough information to choose an operator).

Chunking, the learning mechanism in traditional Soar, can store the results of this planning as a production rule. The production can be used the next time a similar situation is encountered, allowing the agent to perform reactively. However, chunking when there is uncertainty in the Soar agent's model of this environment (as there will most likely be in real environments) can lead to the challenges if an incorrect or undesirable chunk is learned as there is not a simple mechanism for unlearning this chunk.

The mechanism to automatically generate a subgoal to resolve impasses supports the hierarchical execution of complex operators (also referred to as abstract operators). This mechanism allows high-level abstract operators to be implemented in subgoals, in which a new state is created where simpler low-level operators (primitive operators that create actions the Soar agent can perform; e.g., giving motor commands) are used to achieve the goal of the abstract operator [33]. For example, a robotic agent could have an abstract operator with a goal of reaching a specified location. To achieve this goal, it is useful for the agent to consider information such as the relative direction and distance to the location and the presence of

obstacles between the agent and the goal location. The need to consider this additional information makes it difficult for the agent to determine how to perform this operator directly, resulting in an impasse. Within the subgoal generated to resolve the impasse, operators that are able to consider information about where the agent is relative to the location and the presence of obstacles can be used to select the appropriate primitive operators to implement the abstract operator.

Figure 2.3 shows the structure of the traditional Soar architecture. There is one type of long-term (procedural) knowledge (the production rules) and one learning mechanism (chunking). Soar’s decision procedure uses information in the symbolic short-term memory, or working memory, to select operators to generate behavior. Information from processes outside the architecture can be placed in working memory through perceptions and actions can be used to send information from working memory to external processes.

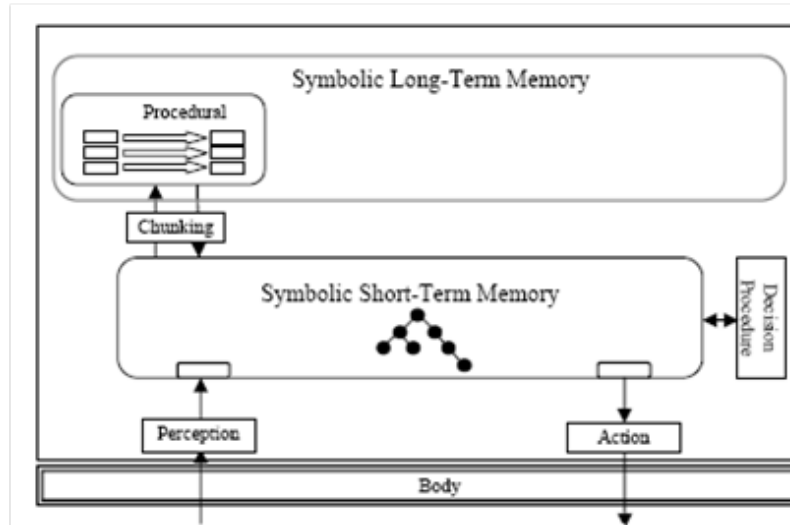


Figure 2.3: Overview of the traditional Soar architecture (Soar 8) [22]. The body at the bottom of the figure refers to interaction between the architecture and a real or simulated external environment.

While the research that will be described in this dissertation uses the mechanisms of the Soar architecture described above, there has been extensive work implementing new mechanisms into the Soar architecture in the last several years [22]. (Soar version 8.6.3 was used for the Soar agent described in Chapter 5 and Soar version 9.3.0 was used for the Soar agent described in Chapter 6, but the new

mechanisms defined in Soar 9 were not used. The most recent Soar release is 9.3.1). These extensions include non-symbolic representations and new types of memories and learning mechanisms that are intended to increase the functionality of Soar (and the “breadth of human behavior that can be modeled using Soar [22]”). New modules that have been, or will be, incorporated into the Soar architecture include semantic memory, episodic memory, reinforcement learning, and visual imagery. Figure 2.4 shows the structure of version 9 of the Soar architecture with its new extensions.

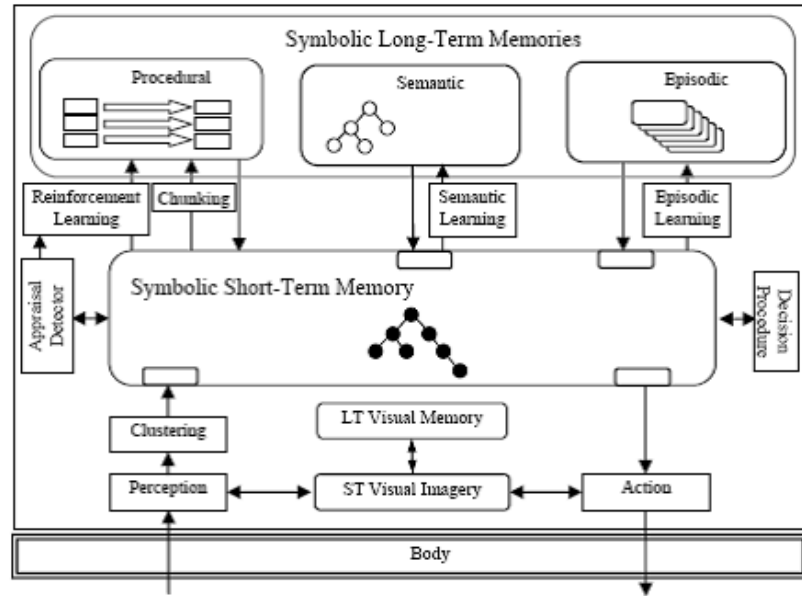


Figure 2.4: Overview of the updated Soar architecture (Soar 9) [22]

Semantic memory is used in Soar to store declarative facts about the world, which are useful for allowing Soar agents to use general knowledge for reasoning [61] or to interact with humans [13]. Episodic memory can be used to remember information about Soar agents’ experiences in the past [62]. Soar’s reinforcement learning module is capable of adjusting the numeric preferences of productions, a parameter that can be used to select operators, based on the potential reward of a production [63]. Visual imagery can be useful for tasks such as spatial reasoning and creating symbolic structures from images and has the potential to simplify tasks that are easier to consider with non-symbolic information [64]. The Soar architecture’s decision cycle is still used to select an operator, but these extensions

can provide additional sources of knowledge that can be used to match production rules and to select among candidate operators.

2.4.2 Soar for unmanned vehicles

This section describes several of the reasons the Soar architecture is desirable for use in unmanned vehicles:

Large knowledge bases: Soar is capable of scaling to the large amount of knowledge needed for intelligent and autonomous behavior in complex, real-world environments, a characteristic which has not been shown by other architectures with a focus on detailed cognitive modeling [13]. Soar uses the Rete algorithm to determine which of the agent’s production rules have their conditions for firing matched based on the current contents of working memory [65]. The use of the Rete algorithm allows sophisticated Soar agents that use detailed knowledge in the forms of rules or sensor information to be developed. For example, as of 1999, an agent in TacAir-Soar used 5200 production rules, 450 total operators, and 130 abstract operators to autonomously fly U.S. military fixed-wing aircraft during missions in a simulated environment [33]. A more recent TacAir-Soar agent has over 8000 production rules [13].

Hierarchical decomposition of operators: The hierarchical decomposition of operators through the use of automatic subgoals is useful for Soar agents, especially those that have a large amount of long-term knowledge. For these agents, even though a large number of operators are likely to be eligible for proposal based on the agent’s current state, only a small number of these operators are likely to be useful for accomplishing the agent’s current goal. Organizing problem solving hierarchically using abstract operators and subgoals allows only operators relevant to the current goal to be considered, limiting the number of operators the agent must consider. Additionally, organizing operators in subgoals can result in “naturally compartmentalized [66]” agents and potentially encouraging reuse of these operators in similar situations.

Mix of reactive and deliberative behavior: Soar agents are able to automat-

ically alternate between reactively selecting operators when there is enough knowledge available to determine the appropriate operator and performing deliberative reasoning to generate enough knowledge to select an operator when reactive selection of an operator is not possible. This deliberative reasoning, including planning, only takes place when necessary, allowing the agent to behave reactively when possible.

Definition of a learning mechanism: Chunking can be used to remember the knowledge created in impasses as a production that can be fired if the agent finds itself in a similar situation in the future, potentially improving performance by allowing the agent to make a decision reactively instead of deliberatively. A challenging aspect of using chunking for unmanned vehicles is that the process requires an internal model of the environment, which can be difficult to obtain accurately in a real environment.

Potential for general intelligence: Since cognitive architectures are good for general decision making, the same architecture can be used across many tasks (along with new specific knowledge for each task). This means a Soar agent could be flexible enough to apply to different mobile robot missions. This agent could also be capable of considering multiple approaches to the same problem and deciding which algorithm would be most effective for the agent's current situation. The agent could switch approaches if the current one is ineffective, leading to the potential for robust behavior.

Truth maintenance system: The use of different types of persistence for working memory elements results in a system for maintaining up-to-date knowledge about the robot's state. Unless the agent deliberately gives a working memory element o-support, the Soar architecture ensures that working memory elements only persist as long as the conditions used in the production that generated them remain true. When these conditions become untrue, the WMEs that relied on the truth of these conditions for their creation are removed. Because of this automated belief maintenance system, Soar programmers do not have to write productions for the agent to maintain accurate state information of i-supported WMEs [66], [67].

Interaction with external systems and environments: The Soar Markup Language (SML) [68] has been developed to allow relatively simple interaction of a Soar agent with external environments through sensors and motors. Since SML can be used with popular programming languages such as Java and C++, integration with other software systems is possible.

Potential for collaboration: The Soar architecture could be used for collaboration between Soar agents and humans as well as between Soar agents and other robots. Instructo-Soar [69] demonstrated the utility of Soar agents that could learn by instruction. Soar agents have demonstrated collaborative behavior (between agents) in simulated external environments. TacAir-Soar agents relied on communication and coordination to perform their flight missions [33]. Another project used Soar agents to control unmanned air and ground vehicles to cooperatively perform tasks such as providing a communications relay, waypoint following, and conducting surveillance during simulated missions [53]. Collaboration between humans and Soar agents controlling robots has also been demonstrated for research focusing on human-robot collaboration for reconnaissance missions [3].

Extensions to Soar: The active development of new mechanisms for the Soar architecture is another compelling reason for using the Soar architecture for the control of unmanned vehicles. The developers of Soar have discussed how these extensions have the potential to benefit unmanned vehicles that use Soar for extended periods of time [47]. Since the Rete algorithm is not designed to scale with the number of working memory elements (as it is with the number of production rules), using the new symbolic memory modules to store information to minimize the growth of working memory can be useful for Soar agents that exist for long periods of time [47]. The wide variety of facts that can be stored in semantic memory module will be important for interactions between Soar agents and humans [13]. While chunking is most useful with an internal model of the environment that can be difficult to acquire in real environments, the new types of learning do not require such a model, a benefit for their use in robots situated in the real world.

The above characteristics suggest that Soar is suitable for high-level control of unmanned vehicles. However, there are also characteristics useful for control of unmanned vehicles that do not match the strengths of Soar. For example, Soar is not well-suited to perform a large amount of numerical calculations, perform low level control of motors, process large amounts of low level information from sensors, or solve optimization problems. Combining Soar with other software systems (including systems such as neural networks [70], robot control algorithms [71], state estimation techniques [72], and object recognition methods such as the Scale Invariant Feature Transform (SIFT) [73]) that complement the strengths of Soar increase the usefulness of Soar for the control of unmanned vehicles.

2.5 Test missions

This dissertation describes the development of the Cognitive Robotic System (CRS) to use the Soar cognitive architecture for the control of unmanned vehicles. The CRS has been applied to two missions that are relevant for unmanned vehicles. The first is autonomous navigation to a goal location in the presence of obstacles and the second is searching for objects of interest in an unknown indoor environment while building a map of the environment. These missions will be described in greater detail in Chapters 5 and 6.

2.5.1 Navigation while avoiding obstacles

The CRS was tested on the mission of navigating to a goal location (specified by a GPS coordinate) in the presence of two types of obstacles. For the first, smaller obstacles were placed such that the robot could move between any two obstacles. This first test is a relatively simple situation that can be accomplished using many approaches. However, this situation can occur frequently and provides an opportunity to confirm the ability of a Soar agent to simultaneously perform the objectives of avoiding obstacles when necessary and moving toward the target location when possible.

The second type of obstacle was more challenging; the goal location was prescribed such that there was a cul-de-sac between the robot's starting position and

the goal location. Ideally, a robot will be able to detect the end of a cul-de-sac before entering it [7]. However, when the end of the cul-de-sac is beyond the limit of the robot's perceptual field and the robot enters the cul-de-sac, the robot must be able to find its way out of the obstacle. This is a commonly encountered problem: avoiding the need to navigate out of a cul-de-sac was considered a primary practical goal of the DARPA LAGR program [7].

Robots that use reactive approaches or remember or consider only a small amount of information about their environment or their past actions may have difficulty escaping from a cul-de-sac [11]. For example, the autonomous navigation software on the Mars Exploration Rovers was updated after the rovers experienced difficulty navigating around complex obstacles such as cul-de-sacs without human intervention [74]. The initial local path planner did not give enough consideration to information about known obstacles near the rovers, relying too heavily on the Euclidean distance from the target waypoint as a metric for selecting a path. The development of the CRS and the results for this mission are described in Chapter 5.

2.5.2 Search in an unknown indoor environment

Conducting a search within an unmapped building for specific objects of interest is another important problem for mobile robots on which the CRS was tested. Perhaps the most studied application of this problem is the use of robots for search and rescue missions. The RoboCup [75] is well known for its soccer competitions, but it also has competitions in the domain of urban search and rescue. The RoboCupRescue competition has divisions for both physical and simulated robots. The simulated division emphasizes advanced robotic capabilities such as formulating plans for multiple heterogeneous agents, learning, and autonomous mission completion, but the physical robots are evaluated on more low level tasks such as map building, identifying simulated victims, and the effectiveness of their human-computer interfaces [76]. The robots compete in three arenas, with each level of arena simulating an increasing level of structural disruption [77]. Teleoperated robots are common, as autonomous robots are allowed to compete only in the least challenging arena. This common use of teleoperated robots is also reflected by the

current practice of robotic surveillance applications [3]. Recently, papers describing more advanced robotic algorithms being applied to search and rescue applications have been written. These papers have included both architecture development for performing search and rescue missions with an attention mechanism to focus the robot's exploration behavior [77] and using an advanced image matching technique to recognize standard hazardous material signs [78].

Important capabilities for successful robots for the problem of search within an unknown building environment include the ability to create a map of the environment, ideally with symbolic information that could easily be shared with humans such as first responders [76], and the ability to identify objects of interest and report where they are located within the environment. These capabilities indicate that this mission requires more high level (symbolic) information about the environment than the first mission of autonomous navigation to a goal location. Developing these capabilities in the CRS make the mission of indoor search a useful challenge for the CRS. The development of the CRS and the results for this mission will be described in Chapter 6.

Hardware

This chapter describes the hardware (robot, processors, and sensors) used in the implementations of the Cognitive Robotic System for the navigation and indoor search missions described in Section 2.5.1. The integration of the hardware components used for these mission will be described in Chapters 5 and 6.

3.1 SuperDroid

The SuperDroid is the wheeled robot used for the research described in this dissertation and is shown in Figure 3.1. The SuperDroid has an aluminum base, which was purchased from SuperDroid Robots [79]. The robot has about 6.7-inch diameter wheels, a width of 21 inches, and a length of 21 inches. The base is strong enough to easily hold a laptop and the other hardware used for this research and its aluminum frame is easy to drill through, allowing for customizable mounting of the hardware to the robot's base. The SuperDroid is driven using four electric motors and a 24V 4000mAh NiMH battery pack, can turn in place, and can be used in challenging outdoor terrain (for example, it can be driven over a curb).

3.2 Processors

There are several processors used on the SuperDroid: a laptop, two microcontrollers, and a motor driver board.

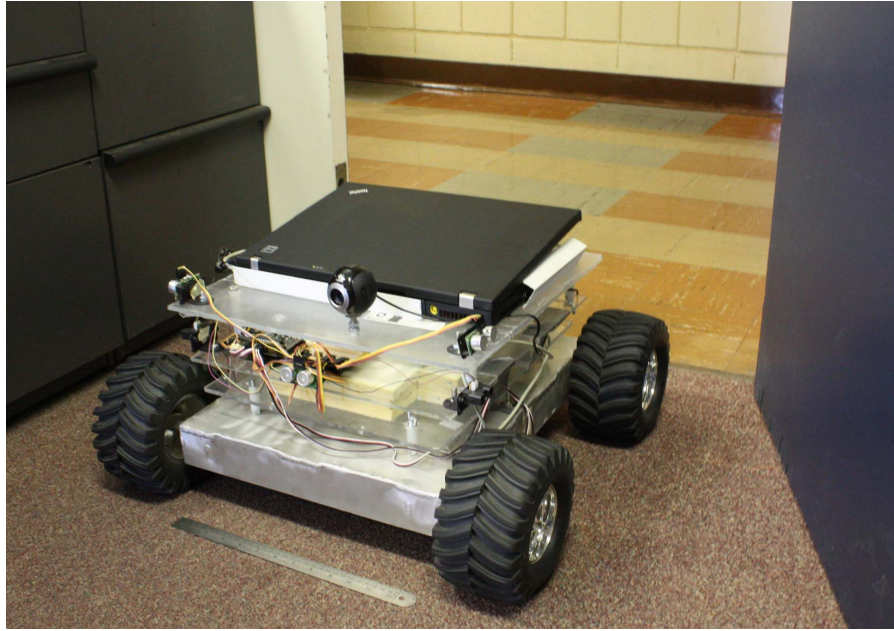


Figure 3.1: A picture of the SuperDroid robot, shown in an office doorway. A 12-inch (about 30 cm) silver ruler is on ground in front of the SuperDroid.

3.2.1 Laptop

A Lenovo ThinkPad with an Intel Core2 Duo 2.0 GHz processor and 2 GB of RAM using the Windows XP operating system is used to communicate with sensors and microcontrollers and runs the Java and Soar programs in the CRS. Several libraries (QTJava.zip, comm.jar, mail.jar, swt.jar, and sml.jar) have been installed for use with the Java language. The Eclipse Integrated Development Environment (IDE) has been installed on the laptop and used to develop and run Java programs. Versions 8.6.3 (for Chapter 5) and 9.3.0 (for Chapter 6) of the Soar architecture is used for the research described in this dissertation.

3.2.2 BrainStem module

A BrainStem module [80] is used to communicate with several sensors, the motor driver board, and the onboard PC. The BrainStem module (Figure 3.2) is a small (2.5-inch by 2.5-inch) microcontroller board that has a 40 MHz RISC processor (the PIC 18C252 microcontroller, which has 1536 bytes of RAM) with multiple input and output capabilities, including servo motor outputs, analog inputs, digital

input/output (I/O) pins, a serial port, and I2C ports. A 6V 2600mAh NiMH battery pack is used to provide power to the BrainStem module. The module is capable of running up to four Tiny Embedded Application (TEA, a subset of the C programming language) programs concurrently.

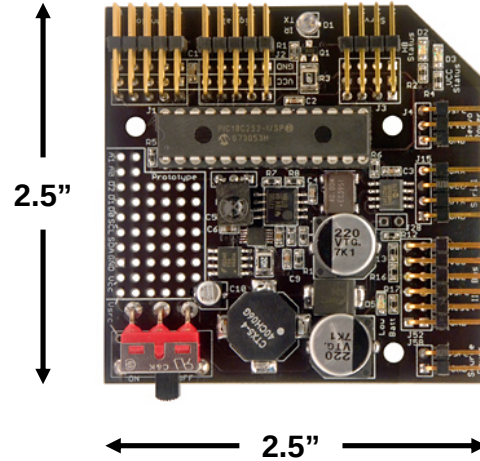


Figure 3.2: A picture of the BrainStem module is shown.

The BrainStem modules are designed to make use of multiple modules simple. The modules can be stacked and each module can act as a router, sending messages from a program on a PC to the BrainStem module for which they were intended using I2C ports, or as a networked module that communicates with a program on a PC by sending and receiving I2C messages through the router BrainStem module.

There are multiple ways in which a BrainStem module can interact with sensors and motors: TEA programs or reflexes can be run on the module or applications on the PC can issue commands to the module. The TEA programs that run on the BrainStem module minimize the amount of low-level sensor interaction that applications on the PC must perform. To send information to applications on a PC, TEA programs can place information (e.g., results from sensors) in an area of memory (called the scratchpad) on the BrainStem that can be accessed by both the TEA programs and applications on the laptop. An example TEA program is provided in Appendix A.

3.2.3 MD22 motor driver

The MD22 Motor Driver (Figure 3.3) from Devantech [81] receives motor commands from a BrainStem module and converts these commands into voltages to drive the SuperDroid's motors. The motor driver uses a 6V 2600mAh NiMH battery pack to provide power for an onboard microprocessor and is also connected to the 24V battery pack used to provide power to the SuperDroid's motors. The MD22 has a mode of operation where it is able to receive a steer command and a throttle command from the servo motor outputs on the BrainStem. Two independent motor commands are sent from the MD22 Motor Driver: one command goes to the two motors on the left side of the SuperDroid while the other command is sent to the two motors on the right side of the SuperDroid. (With an additional MD22 driver, all of the SuperDroid's four motors could be given independent commands.)

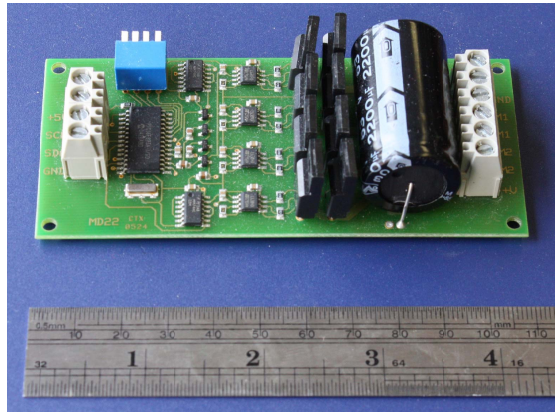


Figure 3.3: A picture of the MD22 motor driver.

3.3 Sensors

This section briefly describes the sensors used for the missions in Chapters 5 and 6. Further details about how the information from each sensor is used will be given in Chapters 4, 5, and 6.

3.3.1 Sonar

The sonar sensor shown in Figure 3.4, the Devantech SRF05 Ultrasonic Ranger [81], can measure the distance to the closest object within the sensor's beam pattern. The sonar sensor can detect obstacles in an approximately fifty degree wide cone shaped field of view at ranges between 3 cm and 4 m. The sensor's measurement is sent to a microcontroller using a digital I/O pins in the form of a digital pulse whose length varies depending on the measurement. On the SuperDroid, a TEA program on a BrainStem module is used to obtain measurements from the sonar sensor and place the result in the BrainStem's scratchpad memory, where it can be accessed by Java programs on the PC.

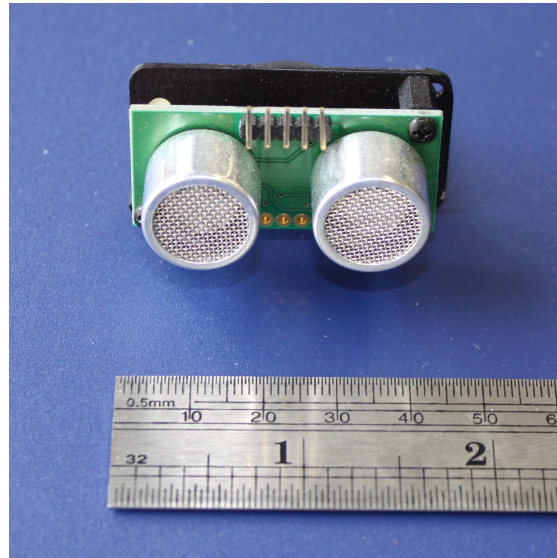


Figure 3.4: A picture of the SRF05 sonar distance sensor.

3.3.2 Infrared

Two models of infrared distance sensors (Figure 3.5) have been used on the SuperDroid. These sensors send an analog voltage proportional to the measured distance to the closest obstacle within the sensor's field of view to an analog input pin on a microprocessor. These infrared sensors detect obstacles in a narrow (about 6-10 cm wide) roughly rectangular area in front of the sensor. The Sharp model number GP2Y0A02YK0F sensor [80] is able to detect obstacles that are between 20 and

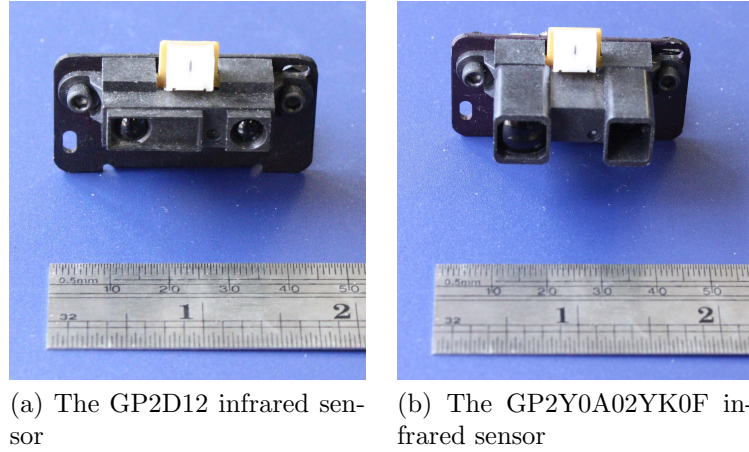


Figure 3.5: Two types of infrared distance sensors

150 cm away from the sensor. The Sharp GP2D12 sensor [80] has a range of 10 to 80 cm. A Java program on the PC is used to send commands to the BrainStem module to get a measurement from the infrared distance sensor by sampling the voltage on the BrainStem's analog input pin to which the sensor is connected.

3.3.3 Compass

The Devantech CMPS03 Electronic Compass [81] shown in Figure 3.6 uses magnetic field sensors to measure the sensor's orientation relative to the Earth's magnetic field with 0.1 degree resolution. The compass's measurements are affected by electromagnetic fields, so the sensor must be placed in a position where the effects of other electronic devices on the compass measurements is minimized. The electronic compass can communicate with an external processor using the I2C protocol or by sending the results as a digital pulse to a digital input pin. Similar to the sonar sensor, a TEA program on a BrainStem module is used to obtain a measurement from the electronic compass and place the result in the BrainStem's scratchpad memory, where it can be accessed by Java programs on the PC.

3.3.4 GPS

The GPS receiver used on the SuperDroid is called the GPS 18 USB [82] (shown in Figure 3.7). The GPS 18 USB is a 12-parallel-channel, WAAS-enabled GPS

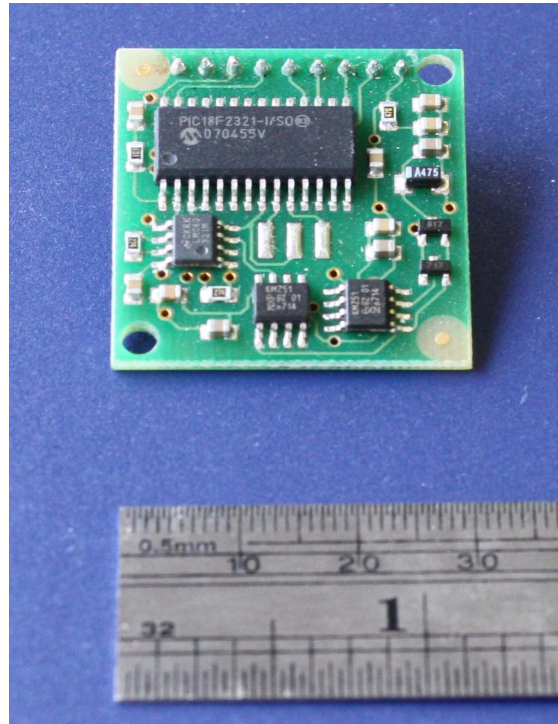


Figure 3.6: A picture of the CMPS03 electronic compass.

receiver that can receive information from up to 12 satellites to update the current latitude and longitude of the receiver. The receiver sends GPS information through a USB port using the standard NMEA sentence format and Garmin’s Spanner interface software (which is installed on the PC).

3.3.5 Wheel encoders

Wheel encoders are commonly used to measure robot movement by monitoring wheel rotation. The incremental quadrature wheel encoders shown in Figure 3.8, the Nubotics WheelWatchers [83], have been installed on the two front wheels of the SuperDroid. This encoder system is designed for RC servos, but has been adapted for use on the Superdroid.

Each encoder attempts to measure the amount of wheel rotation by using two photoreflectors to monitor the rotation of a striped reflective encoder disk (with alternating gray and black stripes) that rotates with the robot wheel. For each front wheel of the SuperDroid, an encoder disk was mounted on a collar that



Figure 3.7: A picture of the GPS unit.

connects the wheel to the front motor. Custom brackets were used to mount the circuit board which holds the two photoreflectors to the SuperDroid frame so the photoreflectors can accurately sense rotation of the encoder disk. Each encoder disk has 32 stripes and since the photoreflectors are placed on the circuit board such that the transitions of the encoder stripes sensed by the first photoreflector are out of phase with the transitions sensed by the second photoreflector, the encoder system is able to provide enough information to sense 128 transitions per wheel revolution as well as the direction of wheel rotation.

Each encoder sends two signals to digital input/output pins on a BrainStem module. A TEA program on the module monitors the two signals from each encoder, keeps track of the current count for each encoder, and places the current encoder counts in the module's scratchpad memory.

3.3.6 Camera

A webcam (the Logitech QuickCam Communicate Deluxe [84], which is shown in Figure 3.9) has been installed on the SuperDroid. The webcam has a 1.3-megapixel sensor and is capable of sending video up to 1280 pixels by 1024 pixels to the PC

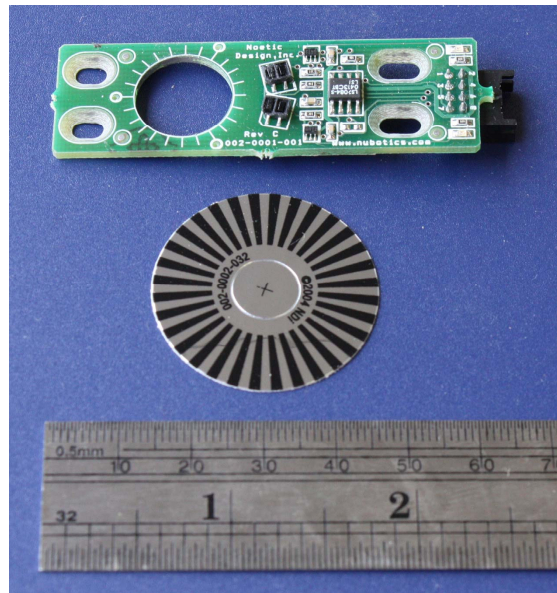


Figure 3.8: A picture of the encoder kit with the circuit board holding two photoreflectors shown at the top and the encoder disk shown in the middle.

using an USB port. Quicktime for Java is used to retrieve pixel information for image processing by Java programs.



Figure 3.9: A picture of the webcam.

Algorithms

Each section in this chapter describes an algorithm that has been implemented in the Cognitive Robotic System (CRS) either to generate information useful for reasoning in Soar or to control motors on the SuperDroid.

4.1 State estimation

The SuperDroid state, x , consists of the robot's x and y positions x_p and y_p and its orientation θ :

$$x = [x_p \ y_p \ \theta]^T \quad (4.1)$$

In indoor environments, this state was estimated using equations that relate information from the encoders (described in Section 3.3.5) to the robot state. Borenstein, Everett, and Feng [85] provide a set of equations, which are summarized here, that can be used for this purpose.

The changes in encoder counts from the last odometry update for the left and right wheels are found by subtracting the previous encoder counts from the current encoder counts:

$$\Delta N_L = N_{L,k} - N_{L,k-1} \quad (4.2)$$

$$\Delta N_R = N_{R,k} - N_{R,k-1}, \quad (4.3)$$

where $N_{s,t}$ is the encoder count of side s at the current time k or the previous time $k - 1$.

The distance traveled by each wheel since the last odometry update was then calculated using a conversion from encoder counts to distance:

$$\Delta U_L = \text{cf} \cdot \Delta N_L \quad (4.4)$$

$$\Delta U_R = \text{cf} \cdot \Delta N_R, \quad (4.5)$$

where cf is calculated according to:

$$\text{cf} = \frac{\pi \cdot d_w}{c_r}. \quad (4.6)$$

Since the robot wheel diameter, d_w , is about 17 cm on the SuperDroid and the number of encoder counts per wheel revolution, c_r , is 132 for the encoders described in Section 3.3.5, cf was calculated to be about 0.004 meters of travel per encoder count.

The distance traveled by the robot since the last odometry update is then estimated by calculating the average of the distance traveled by each wheel:

$$\Delta U = \frac{\Delta U_L + \Delta U_R}{2} \quad (4.7)$$

The change in orientation since the last odometry update is calculated by:

$$\Delta \theta = \frac{\Delta U_R - \Delta U_L}{b}, \quad (4.8)$$

where b is the wheelbase, or distance between the left and right wheels, measured to be about 0.65 m on the SuperDroid.

The robot state at time k is then calculated using the robot state at time $k - 1$ and the distance traveled and change in orientation since the last odometry update:

$$x_{p,k} = x_{p,k-1} + \Delta U \cdot \cos \theta_k \quad (4.9)$$

$$y_{p,k} = y_{p,k-1} + \Delta U \cdot \sin \theta_k \quad (4.10)$$

$$\theta_k = \theta_{k-1} + \Delta \theta \quad (4.11)$$

State estimation using odometry is inexpensive, does not depend on external information or landmarks, and can provide good short-term accuracy [85], but its assumption that wheel rotation equates to linear displacement of the robot on the ground can be easily violated. Slipping wheels (due to slick floors, skidding during turning, or contact between the robot and other objects, for example), unequal wheel diameters, and differences between actual and measured wheel diameters can lead to the accumulation of errors over time as incremental information is integrated. Equation 4.8 also assumes that there is a point contact between the wheels and the ground, but the SuperDroid wheels are almost four inches wide; this may result in additional odometry error. Borenstein et al. [85] described tests that can be used to quantify some types of odometry errors as well as methods that attempt to reduce these errors.

4.2 Motor control

The CRS uses proportional-integral (PI) control to generate the control signals that are sent to the SuperDroid's motors. More details regarding PI control can be found in many textbooks (e.g. [71] and [86]). Given a desired speed, v_d , and angular velocity, ω_d , commands, a PI control program in the CRS uses information from the SuperDroid's wheel encoders to reduce the differences between the desired commands and the motion of the robot.

First, the current speed and angular velocity are calculated using the change in encoder counts for the left and right front wheels:

$$v = \frac{\Delta N_L + \Delta N_R}{2} \quad (4.12)$$

$$\omega = \Delta N_R - \Delta N_L. \quad (4.13)$$

Then the current error for speed and angular velocity are calculated by comparing the measured and desired speed and angular velocity:

$$e_v = v - v_d \quad (4.14)$$

$$e_\omega = \omega - \omega_d. \quad (4.15)$$

The sum of the speed ($\sum_{i=1}^k e_v$) and angular velocity ($\sum_{i=1}^k e_\omega$) errors are updated with the current errors.

$$\sum_{i=1}^k e_v = e_v + \sum_{i=1}^{k-1} e_v \quad (4.16)$$

$$\sum_{i=1}^k e_\omega = e_\omega + \sum_{i=1}^{k-1} e_\omega, \quad (4.17)$$

where k represents the current time.

The control signals for the speed (u_v) and turn (u_ω) are determined using:

$$u_v = k_{P,v} \cdot e_v + k_{I,v} \cdot \sum_{i=1}^k e_v \quad (4.18)$$

$$u_\omega = k_{P,\omega} \cdot e_\omega + k_{I,\omega} \cdot \sum_{i=1}^k e_\omega, \quad (4.19)$$

where $k_{P,v}$ and $k_{P,\omega}$ are the proportional gains for speed and angular velocity and $k_{I,v}$ and $k_{I,\omega}$ are the integral gains for speed and angular velocity. These gains were experimentally tuned.

After a check to ensure the control efforts calculated in Equations (4.18) and (4.19) are not larger than a previously defined threshold, the speed and turn control signals are sent to MD22 driver board.

4.3 Maps for mobile robots

4.3.1 Background

Building a map is a fundamental way in which a mobile robot can generate information and learn about its environment. A robot builds a map by using its perceptual capabilities, such as vision, audio, touch, and range finding, to create a model of its environment. The problem of creating useful representations with noisy sensor data can be very challenging; mapping has been studied in depth. Localization, or the knowledge of where a robot is located with respect to some reference, is important for mapping applications [2]. Often a robot will not know

its location or a map of its environment: the problem of creating a map while localizing a robot with respect to that map at the same time is called Simultaneous Localization and Mapping (SLAM). Thrun has called SLAM one of the “key enabling technologies of mobile robot navigation [87]” and said that this “problem is generally regarded as one of the most important problems in the pursuit of building truly autonomous robots [87].”

Two types of representations have been widely used in robot mapping: metric and topological. There has also been interest in combining these representations in hybrid metric and topological maps.

4.3.1.1 Metric

A metric map is a quantitative description of an environment and can encode locations of obstacles or other objects of interest. Popular forms of metric maps include the occupancy grid [88] and extended Kalman filter (EKF) SLAM [89]. In an occupancy grid, the environment to be mapped is discretized into a grid and each cell in the grid contains information about the probability that it contains an obstacle. Occupancy grids work well for environments with arbitrary, dense obstacles (for example, office environments). However, sharing maps between heterogeneous robots can be difficult, since what one robot considers an obstacle may not be an obstacle to a different robot. Additionally, there are large memory requirements for occupancy grid representations in large-scale environments. In the EKF SLAM algorithm, the map contains information about the locations of features that have been detected in the environment. Common features could include wall corners or edges of door frames. A key drawback of the EKF SLAM algorithm is that it becomes computationally expensive for a large number of map features.

Metric maps (and particularly occupancy grids) are a good match to popular robot sensors: laser and sonar range finders are good at detecting whether space is open or occupied by obstacles. A weakness of using metric representations is that in environments where the robot’s position is not directly observable (through the use of landmarks or an absolute positioning system such as GPS) and the algorithm relies upon robot odometry to estimate the robot’s state, accumulation of odometry errors can result in the accuracy of the metric representation degrading for maps generated over long time frames or for large areas. However, there are ap-

proaches that attempt to minimize the accumulation of odometry errors. Schultz, Adams, and Yamauchi [54] described a continuous localization algorithm that registers a small, short-term occupancy grid with a long-term global occupancy grid to correct for odometry errors while updating the estimate of the robot’s state. Particles filters have been used to take into account uncertainty in a robot’s state while generating maps of Gaussian point landmarks [90] and occupancy grids [91]. Current SLAM methods can produce accurate and useful maps, especially for small environments, but can experience difficulties in environments that are very large or contain loops. While particle filters are popular for closing loops, there is no guarantee that the appropriate hypothesis is represented by a particle [92].

4.3.1.2 Topological

Topological maps generally describe how objects in the environment are connected to each other. Typically, topological maps consist of nodes (landmarks, for examples) that are connected by edges.

Topological representations have the potential to be useful for robot maps: their strengths include areas that are weaknesses of metric representations. Topological maps are better suited than metric maps to compactly represent large-scale environments (Thrun [93] showed that an occupancy grid with 27,000 occupied cells could be transformed into a topological graph of 67 nodes) and perhaps to consider the problem of closing loops as “loop-closing hypotheses are alternative topological structures for the map, not alternative metrical structures [92].”

Additionally, ideas from the area of cognitive mapping have generated interest in the use of topological maps. Research in the area of cognitive maps has generally agreed that topological information including landmarks and paths between these landmarks are important to human navigation [94], [95], providing “ample evidence that humans often use topological information for navigation [87].”

However, there are significant challenges to using topological maps on mobile robots. First, the generation of useful topological maps has faced the symbol grounding problem [96]: being able to reliably abstract “useful symbols from continuous, noisy perceptions of the environment, i.e. how to reliably detect and recognize places and paths [92].”

Another challenge is the correspondence problem: whether landmarks that

appear similar correspond to the same or different nodes in the map. Tomatis stated that “the perception required by a topological approach has to permit the distinction between places [97].” These challenges may help explain why metric maps have been more common in robotics than topological maps. While animals such as insects are able to recognize visual landmarks [?], the most popular types of robot sensors, laser and sonar range finders, are good at identifying whether space is open or not, but have difficulty distinguishing between places. However, if the places represented by nodes in the graph can be distinguished from each other, sharing information between robots or between robots and humans may be easier with topological maps than metric maps.

Two examples of using a previously available topological map to perform navigation are given below: Nourbakhsh [98] described a robot system that was capable of using a previously known topological map to navigate in an office environment. This system generated the state set, a set of possible robot states (the robot’s current possible node locations), and assigned each possible state a certainty factor indicating the likelihood of that state being correct. The robot was capable of using a set of sonar sensors to detect features such as doorways and hallways that could be used to perform localization with respect to the topological map and update the state set and certainty factors. The system was able to successfully generate and execute plans even in the presence of changes to the topological structure of the environments and incorrect detections of features.

Koenig and Simmons [99] described a partially observable Markov decision process (POMDP) used in a robot control architecture for navigation in corridors of an office building. The POMDP accounts for several sources of uncertainty, including actuation, sensing, sensor interpretation, the robot’s initial state, and knowledge about the environment. A POMDP model is generated using a previously obtained topological map of the building. The environment is discretized into spatial locations with sizes of one meter. Each spatial location is represented by four states (each with a different orientation: it is assumed that the corridors in the environment are connected at right angles) that are connected by turning actions. In one example, a topological map with 95 nodes was converted into a POMDP with 3,348 states. The POMDP has different models of states and actions for corridors and for corridor intersections, doorways, rooms and foyers. Included

in the architecture is a process that allows the initial POMDP to be changed as more is learned about the environment’s structure and the models of states and actions included in the POMDP. The POMDP is used to generate and continuously update the probability that the robot is in the states of the POMDP using information about the robot’s actions and sensor measurements. The architecture uses a small, local occupancy grid generated by sonar sensors to generate virtual sensor reports for the areas in front and to both sides of the robot. The robot demonstrated the ability to navigate to a goal location specified by a higher layer of the robot control architecture, even recovering from mistakes such as missing a detection of a corridor or turning to moving down an incorrect corridor.

While the two previous papers described systems that use previously available topological maps, Wang, Chen, and Shi [100] describe a system that is able to generate a topological map. Their system identifies three types of nodes by determining the number of areas for which a laser rangefinder does not sense an obstacle within a predefined range. Once a node has been identified, the robot takes pictures at each node and generates and stores invariant image features (SIFT, which will be described later in Section 4.6) associated with the current node that can be used to determine if the robot has previously been at the current node.

4.3.1.3 Hybrid

Hybrid metric and topological maps for mobile robots attempt to combine the strengths of the two representations. While Section 4.3.1.2 mentioned the difficulty in identifying meaningful landmarks (e.g., intersections) for topological maps, once identified, these nodes can be used to complement the metric representations typically used in robotic maps and have the potential to reduce errors commonly present in metric robot maps (e.g., from odometry drift).

There are many examples of research that use both metric and topological representations. Beeson, Modayil, and Kuipers [92] and Buschka [101] provide thorough reviews of different hybrid map implementations. The papers described in this section illustrate improvements in the ability of hybrid mapping systems to autonomously identify meaningful topological nodes. While earlier approaches have generated topological nodes at set intervals or when specified by a human operator, more recent approaches have used features extracted from sensor infor-

mation [97] or probabilistic metric representations of the local environment [92] to autonomously generate landmarks that are important for navigation decisions.

Yamauchi and Beer [102] described Adaptive Place Networks that modeled uncertain nature of links between topological nodes and was able to adapt its topological representation to changes in the robot’s environment, but the topological nodes were not necessarily created at meaningful places: they were generated after the robot had moved more than a specified from the last node. In this paper, local metric representations were used to reduce the effects of odometry error and to recognize previously visited nodes. Yamauchi et al. [103] have also used Adaptive Place Networks in a system where topological nodes are generated at places specified by a human. The resulting topological map was used to generate a path plan through the topological nodes using a heuristic search algorithm (Dijkstra’s algorithm).

Tomatis argued that since it is easier to maintain global consistency topologically and a robot only “needs to know its precise position with respect to the environment is when it has to interact with environment [97],” a hybrid approach was developed that uses a global topological map and local metric maps for locations where the robot must interact with its environment. This hybrid approach was tested in an office environment with rectilinear corridors. A laser range finder was used to autonomously identify corners and openings (such as intersections or doorways) that are used as both landmarks in the topological maps and as possible reference points for local metric maps. A partially observable Markov decision process (POMDP) was used for topological state estimation and to make decisions about how to move between topological landmarks. When new landmarks (places such as a door or another hallway that lead to a new area) are sensed, they are added to the POMDP as a new state node. Local metric maps are created when a robot entered a room: an extended Kalman filter was used to estimate the locations of lines sensed by the laser rangefinder in that room.

Beeson, et al. [92] present a detailed formal approach, the Hybrid Spatial Semantic Hierarchy or HSSH, for generating a global metric map by using a topological map to generate the global structure of the map and then connecting local metric maps of the topological nodes. Citing the success of metric SLAM methods in small environments without structural ambiguity (loops), a local occupancy

grid was generated using a particle filter to minimize odometry error. This occupancy grid is used for local motion planning and as a local perceptual map that is used to autonomously identify important topological places and create a symbolic description of the local topology (the configuration of the paths entering and leaving) of those places. While these meaningful places are not given labels (such as common types of intersections), the HSSH is not limited to identifying only predefined types of intersections. These places serve as nodes in the global topological map and a snapshot of the local occupancy grid at each node is recorded. The local topologies of nodes are combined to resolve structural ambiguities associated with challenges such as closing loops in the environment. A tree of possible topological maps consistent with the robot's past observations is constructed and the candidate map that best describes the topological structure of the environment is identified. Although the authors mention that a global metric map may not be necessary for many robot activities, the HSSH can also use the global topological map to create a global metric map by connecting the local occupancy grids that have been recorded at all topological nodes. The HSSH has demonstrated the ability to generate an accurate global metric map from global topological information and local metric maps while following a prescribed path in an office environment containing several loops.

There has also been recent interest in the inclusion of semantic knowledge in robotic maps. While landmarks are identified as important or unique points in topological maps, there may not be any useful label associated with these landmarks. Hertzberg and Saffiotti [104] discuss how an ability to assign meaningful labels to these landmarks (such as identifying the type of an intersection) would allow the robot to use these landmarks in a symbolic reasoning system. Examples of projects that have included semantic information in robot maps are given below.

Galindo et al. [105] described a robotic mapping system with parallel spatial and semantic representations for use in navigation tasks in indoor environments. The spatial representation uses both metric and topological maps and is used for planning and execution of robot tasks. In the topological map, nodes represent rooms and corridors and edges represent the connections between nodes. The semantic representation contains information about symbolic concepts useful for indoor navigation. The robot system attempts to establish connections between

symbols in the semantic representation and the corresponding sensor data in the spatial representation. Information in the semantic representation is used for symbolic reasoning such as making inferences that classify rooms based on the objects that have been detected in that room.

Martínez Mozos and Burgard [106] described an algorithm to generate a topological map with semantic information from an occupancy grid of an office environment. A classifier trained using a supervised learning algorithm was used to classify each unoccupied cell of an occupancy grid as a corridor, room, or doorway. The result is a topological map with nodes labeled as corridors, rooms, and doorways: semantic information that has a “direct relation with a human understanding of the environment [106].”

Zender et al. [107] developed a multi-layered spatial representation intended to allow a robot to describe its environment with human spatial concepts to allow for more natural human-robot interaction. The layers in this representation include:

- A metric map that performs SLAM using an extended Kalman filter that uses lines detected by a laser scanner as features.
- A navigation graph that acts as a “roadmap of virtual free-space markers [107].” A node (or marker) is generated when the robot has moved more than specified distance away from another node. Each node in the navigation graph is classified as a doorway, corridor, or room using information from a laser scan taken at that node.
- A topological map is generated by combining sets of connected navigation nodes of the same types to define topological regions that are separated by a navigation node classified as a doorway.
- A conceptual map that supports localization using both geometric (the robot is currently in a room) as well as functional (what objects have been sensed in the current room and what kind of room usually contains sensed objects) information.

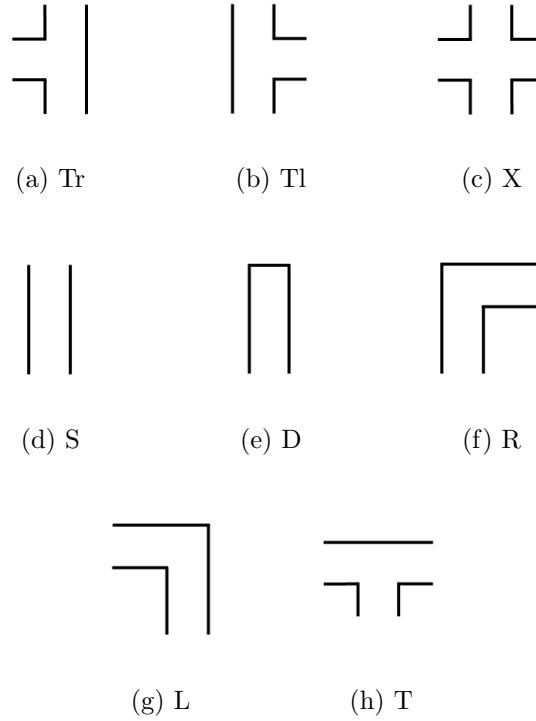


Figure 4.1: Common types of intersections

4.3.2 Maps in the CRS

The primary motivation for including maps in the CRS is to generate symbolic information about the environment that can be used for reasoning by a Soar agent. An occupancy grid algorithm has been implemented to provide a framework to fuse information from different sensors as well as integrate sensor data over time and create a local metric representation of the agent's environment. While occupancy grids are commonly used for tasks such as path planning and obstacle avoidance, it is also possible to extract symbolic information from occupancy grids. Occupancy grids will be used to detect types of intersections (shown in figure 4.1) commonly found in buildings. These detected intersections will be used as nodes in a topological map created by a Soar agent to use for exploration inside a building.

4.3.2.1 Occupancy Grid Algorithm

The problem of creating occupancy grids has been well studied, so not all the details are included here. A more detailed derivation is available in chapters four

and nine of *Probabilistic Robotics* by Thrun, Burgard, and Fox [108]. The goal of a mapping algorithm is to estimate the probability, or likelihood, of a map, m , given the measurements for times 1 through t , $z_{1:t}$, and robot states for times 1 through t , $x_{1:t}$:

$$p(m|z_{1:t}, x_{1:t}). \quad (4.20)$$

The occupancy grid approach divides the environment into cells:

$$m = m_i. \quad (4.21)$$

The probability that the i^{th} grid cell is occupied (i.e. the probability that the grid cell has a value of 1) is denoted by: $p(m_i = 1)$ (or $p(m_i)$). Generally, occupancy grid algorithms assume that the probability of each grid cell m_i being occupied can be considered independently and that the map is static. With these assumptions, the map can be calculated by calculating the probability that each grid cell m_i is occupied, given the measurements for times 1 through t , $z_{1:t}$, and robot states for times 1 through t , $x_{1:t}$:

$$p(m_i|z_{1:t}, x_{1:t}). \quad (4.22)$$

For computational reasons, the log odds ratio of Eq. (4.22) is often used for implementations of occupancy grids. The odds ratio is defined as:

$$\text{Odds}(m_i) = \frac{p(m_i|z_{1:t}, x_{1:t})}{p(\neg m_i|z_{1:t}, x_{1:t})} = \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})}. \quad (4.23)$$

Then, taking the log of both sides of Eq. (4.23) results in:

$$\log(\text{Odds}(m_i)) = l(m_i) = \log \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})}. \quad (4.24)$$

Several sources have shown how Eq. (4.24) can be used to calculate $l_{t,i}$, the log odds ratio of cell i at time t :

$$l_{t,i} = \log \frac{p(m_i|z_t, x_t)}{1 - p(m_i|z_t, x_t)} + \log \frac{p(m_i|z_{1:t-1}, x_{1:t-1})}{1 - p(m_i|z_{1:t-1}, x_{1:t-1})} + \log \frac{p(m_i)}{1 - p(m_i)}. \quad (4.25)$$

The term $p(m_i|z_t, x_t)$ in Eq. (4.25) is the inverse sensor model, which calculates the probability that a grid cell i is occupied given the current (at time t) measurements and robot state.

If nothing is known about the map of the environment at $t = 0$, as will be the case in our mission, $p(m_i) = 0.5$ and the last term in Eq. (4.25) will be equal to zero. Also, noting that:

$$\log \frac{p(m_i|z_{1:t-1}, x_{1:t-1})}{1 - p(m_i|z_{1:t-1}, x_{1:t-1})} = l_{t-1,i}, \quad (4.26)$$

where $l_{t-1,i}$ is the the log odds ratio of cell i at time $t - 1$, Eq. (4.25) can be simplified to:

$$l_{t,i} = \log \frac{p(m_i|z_t, x_t)}{1 - p(m_i|z_t, x_t)} + l_{t-1,i}. \quad (4.27)$$

The probability that a cell i is occupied can be calculated from the log odds ratio for that cell:

$$p(m_i|z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 + \exp(l_{t,i})}. \quad (4.28)$$

4.3.2.2 Inverse Sensor Models

An inverse sensor model must be defined for each type of sensor used to generate the occupancy grid. The inverse sensor model in its log odds form is given by: $\log \frac{p(m_i|z_t, x_t)}{1 - p(m_i|z_t, x_t)}$ from Eq. (4.27). Thrun et al. [108] gave an overview of a simple inverse sensor model for range finding sensors similar to those that have been used for the occupancy grid algorithm in the CRS. This simple model increases the probability that the grid cells within the sensor beam that are approximately the same distance from the sensor as the measured range are occupied. The model decreases the probability that the grid cells within the sensor beam that are closer to the sensor than the measured range are occupied. For cells which are outside the sensor beam or farther away from the sensor than the measured range, the model returns the same probability as before the measurement.

Since both sonar and infrared distance sensors are used to create the occupancy grid, an inverse sensor model must be defined for each type of sensor. The inverse sensor models used for the sonar and infrared sensors differ only by the cells up-

dated by the model (determined by the perceptual field of view of each sensor) and the value of the parameters l_{free} and l_{occ} . The model is described below and calculated for each occupancy grid cell m_i within the perceptual field of view of each sensor.

- First, r_i , the distance between the current cell m_i and the sensor is calculated.
- If the distance r_i is less than the current sensor measurement, z_i , then the current cell m_i is likely to be unoccupied, and the inverse sensor model returns a value l_{free} that will decrease the probability that the cell m_i is occupied.
- If the distance r_i is greater than or equal to the current sensor measurement, z_i , and the difference between r_i and z_i is less than the expected obstacle thickness t , the current cell is likely to be occupied by an obstacle and the inverse sensor model returns a value l_{occ} that will increase the probability that the cell m_i is occupied.
- If the distance r_i is greater than or equal to the current sensor measurement, z_i , and the difference between r_i and z_i is greater than the expected obstacle thickness t , the current sensor reading does not give any new information about the contents of the current cell and the inverse sensor model returns a value l_0 that will not change the probability that the cell m_i is occupied.

Figure 4.2 shows example results from the inverse sensor models for the two sensors used to generate the occupancy grids in the CRS. More sophisticated inverse sensor models are available: for example, some models use statistical distributions to weigh cells in the middle of the sensor's perceptual field more than cells at the edge of the perceptual field. A more complex inverse sensor model could be implemented if the performance of this simple sensor model is inadequate. Figure 4.2 illustrates the complementary features of the sensors. The sonar sensors are good for sensing free space, but can not tell where the closest obstacle it senses is located within its wide perceptual field. The infrared sensors have a much narrower perceptual field and are useful for sensing openings, such as doors, that can be more narrow than the sonar sensor's perceptual field.

By using different values of the parameters l_{free} and l_{occ} , the measurements from the sonar and infrared sensors can be weighted differently. This also allows

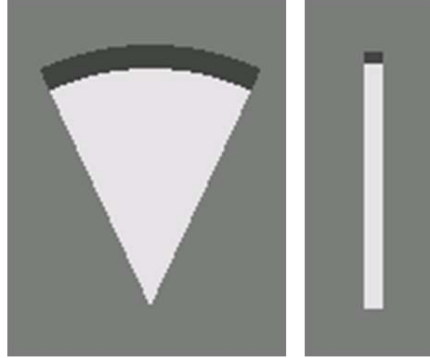


Figure 4.2: A comparison of sonar (left) and IR (right) perceptual fields.

different measurements from the infrared sensors to be weighted differently: when an infrared sensor returns a measurement equal to its maximum sensor range, this measurement is weighted heavily because it is possible that the measurement represents information that a sonar sensor is not capable of measuring. For example, while a sonar sensor facing a door may return a measurement of the distance to one side of the door because its perceptual field is wider than the door opening, an infrared sensor facing the same door would likely return a measurement equal to its maximum sensor range since its perceptual field is narrower than the door opening. However, if an infrared measurement senses an obstacle at a distance less than its maximum range, this measurement is weighted less heavily than measurements from a sonar sensor because this measurement is likely to be duplicating information from a sonar sensor and infrared measurements have higher standard deviations than sonar measurements.

4.3.3 Implementation and Results

The occupancy grid algorithm described in Section 4.3.2.1 was implemented within the CRS using Java. Three sonar and two infrared distance sensors were used to detect obstacles in the SuperDroid's environment. The sonar sensors were placed to detect obstacles to the left, in front of, and to the right of the robot. The infrared sensor were placed to detect obstacles to the sides of the robot. The robot's state was estimated using information from the wheel encoders, as described in Section 4.1. Figure 4.3 shows the occupancy grid generated using five measurements from each of the sonar and infrared sensors while the robot is stationary in a hall with

walls to its left and right and free space in front of the robot. This figure shows the areas where the robot's sensors do not provide information about the environment (the gaps in information in front of and to the sides of the robot). The perceptual field of the sonar sensor in front of the robot can also be seen: this sensor is useful for detecting the presence of an obstacle in front of the robot, but has limited ability to localize that obstacle and is not useful for avoiding small obstacles or differentiating between small and large obstacles.



Figure 4.3: Occupancy grid generated by a stationary robot.

Since the motivation for using the occupancy grid is to integrate sensor information over time to identify intersections and not to create a metric map of the entire environment, the CRS uses a local occupancy grid that is larger than the expected sizes of the intersections it is trying to detect. The local occupancy grid moves with the SuperDroid as the robot moves through its environment, similar to the system described by Beeson et al. [92]. The occupancy grid is translated to keep the robot's current state near the center of the grid. To avoid information in grid cells being blurred into several cells, the occupancy grid is not rotated (the robot's orientation with respect to the grid is allowed to change) and is always translated by an integer number of cells. Since the occupancy grid used by the CRS represents a 4 meter by 4 meter region, odometry errors should be minimal under normal circumstances.

To demonstrate the importance of using both sonar and infrared distance sensors to generate occupancy grids for the current sensor configuration, occupancy grids were generated while the SuperDroid navigated down a hallway with an open

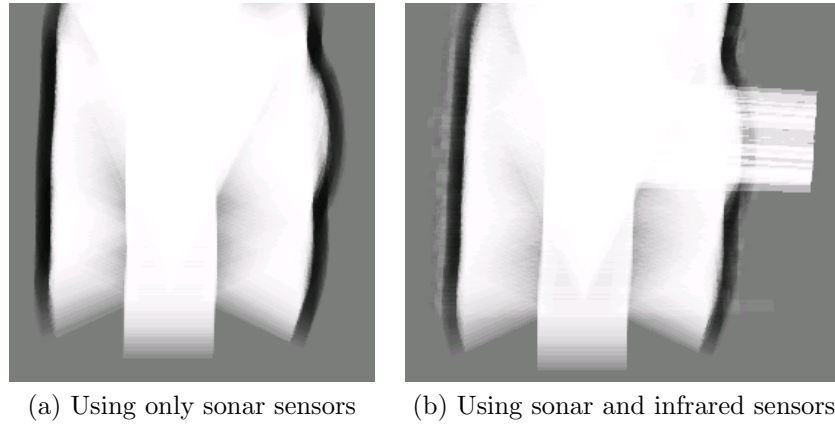


Figure 4.4: Occupancy grid maps of a hallway with an open office door in the right wall

narrow door to the right. The SuperDroid navigated down the middle of the hallway with its right sonar and infrared sensors approximately 70 cm from the right wall and open door. Figure 4.4a shows the occupancy grid generated using only the three sonar sensors on the SuperDroid. The black vertical regions represent the walls of the hallway. The slight bump to the right on the right wall is at the location of the open door. Figure 4.4b shows the occupancy grid generated using the three sonar sensors and the two infrared sensors on the SuperDroid. In this occupancy grid, the infrared distance sensor on the right of the SuperDroid was able to sense the free space associated with the door and the open door in the right wall is clearly seen. It is also possible to see the effects of noisy infrared distance measurements near the left wall of Fig. 4.4b.

Figure 4.5 shows occupancy grids generated while the robot was in each of the eight common intersection types shown in Fig. 4.1. The occupancy grids generated by the CRS are able to show the presence or absence of features such as walls that will be used to determine the intersection types that may be present in the grid. Although the building in which these maps were generated only contained hallways connected at right angles, there are a variety of hall orientations depicted in these occupancy grids. When the robot first starts navigating through a building, its orientation is set such that it is moving towards the top of its occupancy grid (similar to the situation shown in Figure 4.5b. With perfect odometry information, all hallways should be oriented vertically or horizontally in the occupancy grids shown

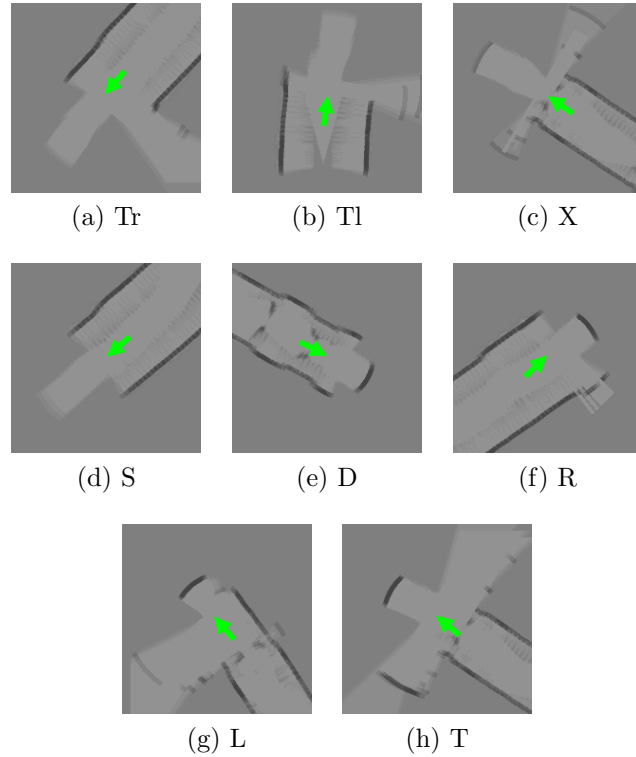


Figure 4.5: Occupancy grid maps of different intersection types. The green arrows represent the position and orientation of the robot on each occupancy grid.

in Figure 4.5 since they were generated in a building in which all halls intersect at right angles. The deviation of the hallways' orientations from vertical and horizontal in these figures indicate significant odometry error has accumulated over time. However, the walls shown in these occupancy grids are relatively straight, indicating that odometry errors are not large within the small, local occupancy grids. While map algorithms that take into account uncertainty in robot position (for example, FastSLAM [90] or continuous localization [54]) could have been implemented in the CRS, the main use of the occupancy grid is as an input to a system that can detect common types of intersections and this system has been developed to function without a globally accurate estimation of the robot state.

4.4 Hough transform

4.4.1 Background

The Hough transform [109] is a popular technique for identifying the dominant lines in a binary image. Generally, to identify these lines in an image, an edge detection algorithm is used to generate a binary image of edge pixels. Most applications of the Hough transform for identifying lines use the fact that a line in an image can be represented in polar form by the equation:

$$\rho = x \cos \theta + y \sin \theta, \quad (4.29)$$

where ρ is the normal distance from the image origin to the line, θ is the orientation of the normal line, and x and y are pixel coordinates.

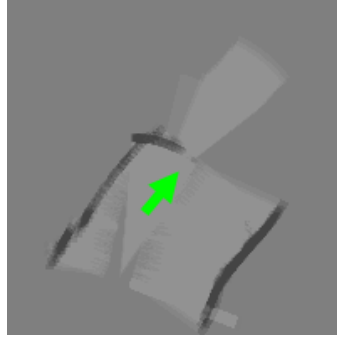
The Hough transform creates a two-dimensional discretization of the ρ - θ parameter space where each ρ - θ pair represents a possible line that can be identified by the transform. A two dimensional array is used to store the number of pixels that could be part of the line represented by each ρ - θ pair. It is possible that each binary edge pixel could be part of many lines represented by unique ρ - θ pairs. For each edge pixel, the number of pixels associated with each line (ρ - θ pair) that the current edge pixel could belong to is incremented. After each edge pixel has been considered, the ρ - θ pair that has the most possible pixels is considered to be the most dominant line in the image. Since edge pixels do not have to be contiguous to belong to the same line, the Hough transform can identify a dominant line even if part of the line is occluded in the image. The Hough transform is widely used and is described in computer vision textbooks (e.g., Trucco and Verri [110]).

4.4.2 Implementation and results

In the CRS, Hough transforms are used to identify lines that are likely to correspond to walls in an occupancy grid. Similarly, Yamauchi has described using a Hough transform for identifying building walls from laser range data in an urban reconnaissance mission [111]. In the CRS, the walls are then used to estimate the orientation and width of the robot's current hall. Figure 4.6 shows the results of the process used to identify the locations of the walls in an example occupancy grid

(Figure 4.6a). First, all cells in the current occupancy grid that have a probability of containing an obstacle that is above a specified threshold are given a value of one and the remaining cells (those not likely to contain an obstacle) are given a value of zero. The result is a binary grid of obstacles, shown in Figure 4.6b.

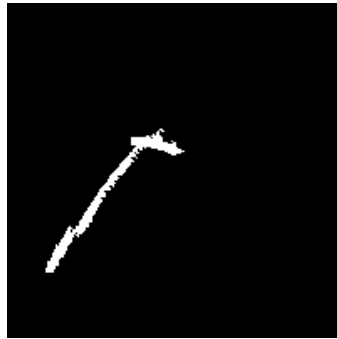
Using information about the robot's current location and orientation, the binary obstacle grid is divided into two images: one that contains the cells with obstacles in the occupancy grid to the left of the robot (Figure 4.6c) and a second that contains the cells with obstacles in the grid to the robot's right (Figure 4.6d). A hough transform is then performed on both images to identify the dominant line in each binary obstacle grid. The number of contributing pixels for the dominant line in each image are compared and the line with the higher number of contributing pixels is identified as the most dominant line and the location of that line (to the left or right of the robot) is noted. For this example, the line to the left of the robot (Figure 4.6e) has the highest number of contributing pixels and is denoted as the most dominant line in either binary obstacle grid. Once the side of the most dominant line has been identified, the algorithm looks in the obstacle grid on the other side of the robot for the set of lines that have similar orientations as the most dominant line and identifies the line in this set with the highest number of contributing pixels (Figure 4.6f). Once these two lines are identified, the orientation of each line is transformed into the coordinate system in which the robot's orientation is described. Additionally, the ρ values of the two Hough lines are used to estimate the width of the robot's current hall by calculating the distance between the two lines identified in Figures 4.6e and 4.6f.



(a) An occupancy grid, with a green arrow to represent the robot's location and orientation



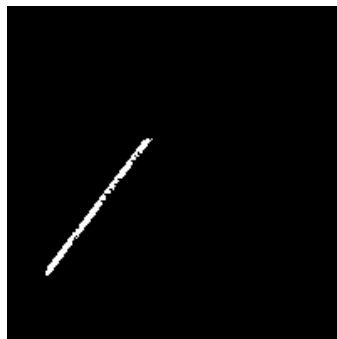
(b) The binary obstacle grid, with cells containing obstacles in white



(c) Obstacles located to the left of the robot



(d) Obstacles located to the right of the robot



(e) Dominant line, corresponding to left wall of occupancy grid



(f) Most dominant line to right of robot with a similar orientation to dominant line to left of robot

Figure 4.6: Hough transform results

4.5 Intersection Detection

Once occupancy grid maps are generated in the CRS, they are used to detect common types of intersections. This section describes how similar problems have been approached in the past and describes the current solution in the CRS.

4.5.1 Generating semantic spatial information

Section 4.3.1.3 discussed research that has generated maps that have included semantic information in their representations. This section describes in more depth some of the procedures used in these projects to extract semantic information from the environment. Classifiers, symbolic rules, neural networks, and image processing techniques have all been used to generate information with meaningful labels from occupancy grids or sensor data.

As described in Section 4.3.1.3, Martínez Mozos and Burgard [106] extracted semantic information from an occupancy grid by using a previously trained classifier to recognize unoccupied occupancy grid cells as corridors, rooms, and doorways. A virtual scan was performed in each unoccupied cell of an occupancy grid and simple geometric features were calculated from the 360 virtual laser measurements. A boosting algorithm used these features as a set of weak classifiers to create a strong classifier that could accurately classify each unoccupied cell. After classification, unoccupied cells were combined into regions, which were then used to generate a topological map.

Galindo et al. [105] used image processing techniques to identify large open spaces in the occupancy grid that corresponded to rooms and corridors. An occupancy grid was treated as a grayscale image and morphologically filtered and then segmented into topological regions. Additional image processing operators were applied to these regions to calculate parameters such as the eccentricity of the region and the width and height of a box that bounds the region that were used to classify each region as a room or corridor [112].

Kelley, Avery, Long, and Dimperio [26] used a laser rangefinder to identify seven types of intersections in the SS-RICS architecture. Rangefinder measurements of three points (in front and to the left and right of the robot) were compared to a threshold and each of the three points was concluded to be open or blocked.

Once the paths in front of and to the left and right of the robot were classified as open or blocked, SS-RICS used symbolic rules to determine the presence of an intersection. The difficulty in selecting the appropriate thresholds to use for correctly identifying an intersection was discussed. The authors also mentioned a desire to use neural networks (including spiking neural networks [113], [114]) for intersection identification in the future.

4.5.2 Implementation

A fuzzy logic algorithm has been developed that considers information from the occupancy grid and attempts to detect the eight types of intersections from Fig. 4.1. The details of this approach are discussed in the following sections.

4.5.2.1 Virtual rangefinder

A virtual rangefinder was developed that uses the binary obstacle grid generated from the occupancy grid (as described in Section 4.4) as input to a virtual sensor. The virtual rangefinder is placed at the center of the obstacle grid and the entire circular region around the rangefinder is divided into 180 arc regions. Within each of the 180 arc regions, the closest cell that contains an obstacle is located and the distance from the simulated rangefinder to this cell is calculated. If no obstacle is sensed within the region, a maximum measurement of 3m is returned for that region. Therefore, each virtual scan results in 180 range measurements, each of which represents the distance to the closest obstacle from the rangefinder in that arc region.

The 180 arc regions are separated into 12 groups, each consisting of 15 arc regions, (depicted in Figure 4.7) and an average range is calculated for each of the 12 groups. The average range in three areas: straight ahead of and to the left and right of the rangefinder's position on the occupancy grid are used as input to the fuzzy logic system.

4.5.2.2 Fuzzy logic system

Fuzzy logic provides a mechanism to represent uncertainty that is a result of “the absence of sharply defined criteria of class membership rather than the presence of

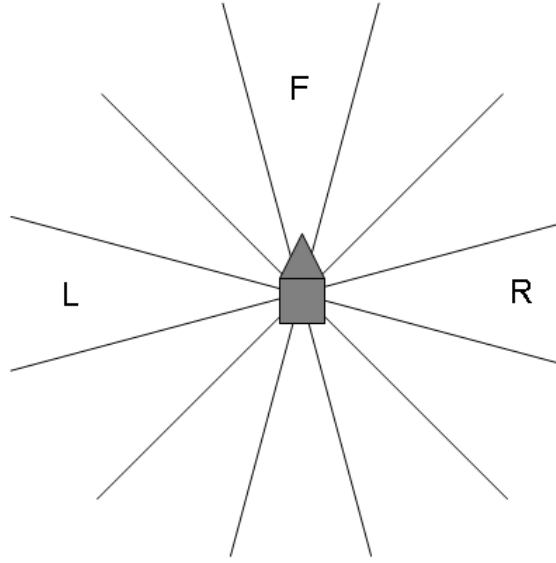


Figure 4.7: The 12 groups for which the average rangefinder distances are calculated are shown. The rangefinder is facing the top of the figure and ‘L,’ ‘F,’ and ‘R’ represent that left, front, and right groups.

random variables [115].” In other words, fuzzy logic can represent vagueness or a lack of precision instead of randomness as in probability theory [116]. This mechanism can be useful for complex systems whose behavior is not well understood or for systems where exact solutions are not necessary [117].

A fuzzy set contains elements that have varying degrees of membership to the set. In classical logic systems a variable either does or does not belong to a set. However, in fuzzy logic systems, the degree to which a variable belongs to a set is described by a real number between 0 and 1. The degree of membership can also be referred to as the certainty or confidence value that the element belongs to the set. The certainty that an element or variable belongs to a fuzzy set is calculated using a set membership function. Fuzzy set theory also defines operators that determine confidence values for expressions that contain fuzzy variables. The three standard fuzzy operations are union, intersection, and complement [117].

The CRS uses the fuzzy logic system to generate a confidence value that the robot is located in the intersection types shown in Figure 4.1. The system uses three fuzzy sets (`wallOnLeft`, `wallOnRight`, and `wallInFront`) that represent the

degree to which a wall is present to the left, in front, or to the right of the robot. Membership functions are defined to calculate the certainty that the results of the virtual rangefinder belong to these sets by using the average ranges in the regions to the left, in front, or to the right of the robot and the width of the current hall (estimated using the Hough transform described in Section 4.4). Using the estimated hall width as a parameter in the membership functions allows the intersection detection system to adapt to different size halls. Figure 4.8 shows the membership function used to determine the certainty that a wall is present to the left of the robot in a two meter wide hall.

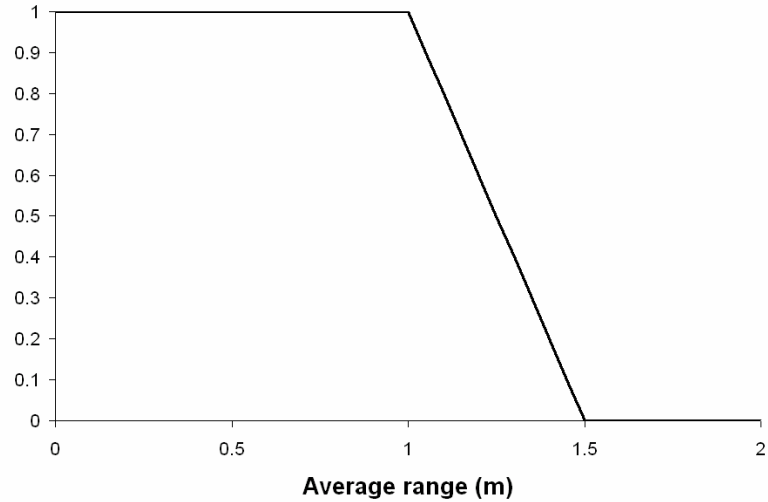


Figure 4.8: An example membership function to calculate the certainty that there is a wall to the left of the robot.

Once membership functions have calculated the certainty that the current virtual rangefinder results belong to the wallOnLeft (L), wallOnRight (R), and wallInFront (F) sets, fuzzy intersection and complement operators are used to determine the confidence values the robot is located in each of the common intersection types. The confidence value of the intersection of fuzzy sets A and B is denoted by $\mu_{A \cap B}(x) = \mu_A(x) \wedge \mu_B(x)$ and the confidence value of the complement of fuzzy set A is denoted by $\mu_{A'} = 1 - \mu_A(x)$.

The confidence values of the presence of each intersection type are represented

by $\mu_n(x)$ where n is the intersection type and x is the current virtual rangefinder result and are calculated using Equations 4.30–4.37.

$$\mu_D(x) = \mu_L(x) \wedge \mu_F(x) \wedge \mu_R(x) \quad (4.30)$$

$$\mu_S(x) = \mu_L(x) \wedge \mu_{F'}(x) \wedge \mu_R(x) \quad (4.31)$$

$$\mu_R(x) = \mu_L(x) \wedge \mu_F(x) \wedge \mu_{R'}(x) \quad (4.32)$$

$$\mu_L(x) = \mu_{L'}(x) \wedge \mu_F(x) \wedge \mu_R(x) \quad (4.33)$$

$$\mu_T(x) = \mu_{L'}(x) \wedge \mu_F(x) \wedge \mu_{R'}(x) \quad (4.34)$$

$$\mu_{T_L}(x) = \mu_L(x) \wedge \mu_{F'}(x) \wedge \mu_{R'}(x) \quad (4.35)$$

$$\mu_{T_R}(x) = \mu_{L'}(x) \wedge \mu_{F'}(x) \wedge \mu_R(x) \quad (4.36)$$

$$\mu_X(x) = \mu_{L'}(x) \wedge \mu_{F'}(x) \wedge \mu_{R'}(x) \quad (4.37)$$

4.5.3 Results

Results of using the fuzzy logic algorithm for intersection detection for each of the occupancy grids shown in Figure 4.5 are given in Table 4.1. Each column of Table 4.1 shows an occupancy grid from Figure 4.5 and the confidence value that each type of intersection is present in that occupancy grid. The column displaying the results for the ‘D’ intersection shows that the sum of the confidence values for each occupancy grid does not have to add to one, as would be necessary for a probabilistic framework for estimating the likelihood of the presence of common intersection types. Even in the presence of noisy occupancy grids in which it is difficult to localize obstacles in front of the robot, the fuzzy logic algorithm is able to assign the correct type of intersection the highest confidence value for each occupancy grid.

4.6 SIFT

Image matching is an important capability for many computer vision problems, including object recognition, scene recognition, and visual odometry. Image features that can be found repeatedly under different image conditions can be very

Table 4.1: Confidence values for the presence of common intersection types, detected from occupancy grids

		Occupancy Grids							
									
Intersection types	Tr	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Tl	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
	X	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.15
	S	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
	D	0.00	0.00	0.00	0.00	0.81	0.00	0.00	0.00
	R	0.00	0.00	0.00	0.00	0.18	0.98	0.00	0.00
	L	0.00	0.00	0.00	0.00	0.09	0.00	0.94	0.00
	T	0.00	0.00	0.00	0.00	0.09	0.02	0.06	0.85

useful for image matching.

4.6.1 SIFT Background

The Scale Invariant Feature Transform (SIFT) [73] is a computer vision algorithm that has been widely used for image matching. SIFT detects local image features and creates descriptions of the region around these features that can be used to match the features between different views of an object or scene containing the features, even in the presence of scaling, rotation, occlusion, and small variations in “illumination and 3D camera viewpoint [73].” There are several characteristics of SIFT features that make them useful for reliable matching. In addition to being invariant to image scaling and rotation, the features are also highly distinctive, which permits a single feature to be correctly matched in a large database of features from multiple images. Lowe showed that SIFT features could be used to recognize previously seen objects in cluttered images. First, SIFT features were identified in a training image of each object. The location, orientation, and scale of each feature were recorded and a descriptor vector (also called the SIFT feature vector) was created to characterize the image area around each feature. The feature information and feature vector were placed in a database of SIFT features. Next, a test image containing the objects was considered. Again the SIFT features of the image were identified, the feature location, orientation, and scale were recorded,

and the feature vector was created for each feature. To perform the matching, the feature vectors from the test image were compared to the database of feature vectors from the training image. Close matches were checked for geometric consistency using the feature locations, orientations, and scales.

While SIFT has been found to perform quite well compared to other local features [118], there have also been attempts to replicate the functionality of SIFT in other algorithms such as SURF (Speeded-Up Robust Features) [119]. Bauer et al. [120] compared the ability of several implementations of the SIFT and SURF detectors to generate matches in the presence of differing image conditions. The authors reported that using SIFT features generally resulted in both a slightly higher number of correct matches and a higher ratio of correct to incorrect matches than using SURF features, but that SURF was able to perform almost as well in a shorter amount of time.

4.6.2 Applications of SIFT

SIFT has been widely used in the computer vision and mobile robot communities. In addition to demonstrating recognition of previously seen objects amid clutter even when the objects are partially occluded, Lowe also showed that SIFT features can be used to recognize a familiar location [73].

SIFT features have also been used for one of the most challenging aspects of SLAM: the data correspondence problem. The data correspondence problem refers to the challenge of distinguishing similar objects in the environment. For example, a robot may find several vertical edges in an image. However, after it moves, the locations of these edges in the image will have moved and it can be very difficult to know which edge in the new image corresponds to that edge in the old image. Se, Low, and Little [121] matched SIFT features in successive images taken from a camera on a mobile robot to correct the odometry drift experienced by that robot. Se, Lowe, and Little [122] have also developed a system that uses SIFT features to globally localize a robot, as in the kidnapped robot problem. The usefulness of SIFT features for closing loops was also explored.

SIFT features do not necessarily correspond to locations within an image that would be meaningful to a human, as the features are selected so that they can be

identified in a repeatable manner. Researchers such as Folkesson and Christensen [123] have attempted to use other image features that may be more meaningful (such as corners) in conjunction with the SIFT feature vector. Also, as described earlier in Section 4.3.1.2, Wang et al. [100] developed a system that generated SIFT features that are associated with identified instances of common types of intersections. A benefit of associating SIFT features only with meaningful locations as described (and not with all locations) in these papers is the reduction in the total number of SIFT features that must be generated. In addition to taking less memory to store all the SIFT features, there are fewer locations and corresponding SIFT features that must be considered for matching the current location.

SIFT features have also been used on mobile robots to locate objects of interest in an indoor environment. Meger et al. [124] described a system that maps its environment using an occupancy grid and appends the map with locations of objects found using SIFT features. Objects such as a basketball, recycling bin, a box of detergent, and a photograph are identified by comparing SIFT features from training images of these objects to images captured using an onboard camera. Similarly, Zender et al. [107] used SIFT features for recognition of objects for which training images are available in a mobile robot system (described previously in Section 4.3.1.3) that searches an indoor environment.

4.6.3 SIFT in the CRS

The CRS uses SIFT features to recognize previously seen objects. An executable file created by the developer of SIFT is available for download at the developer's website [125]. The executable file for Windows (win32sift.exe) takes an image file as input, locates the SIFT features in the image, generates the SIFT feature vector for each SIFT feature, and outputs the features and feature vectors to a text file.

This executable has been incorporated into the CRS and code has been developed to compare SIFT features between two images to detect SIFT features that are present in both images. The grayscale pixel values of an image, either from a webcam or image file, are sent to the SIFT executable and the SIFT features and their feature vectors that are output from the executable are stored.

The CRS uses a previously saved image of an object (or images of the objects)

to be recognized and generates a set of SIFT features, called the training features, associated with that object. To look for the object, an image is taken using the webcam on the robot and SIFT features (called the test features) are found using the SIFT executable. The training and test features are then compared to determine if the training object is present in the image from the webcam. The Euclidean distances between each test feature and each training feature are calculated. The two training SIFT features that most closely match each test SIFT feature are identified and the ratio of the distance for the closest match to the distance for the second closest match is calculated. If the ratio is smaller than a threshold, indicating that the match between the test feature and the closest training feature is significantly better than the match between the test feature and the second closest training feature, the test feature is considered to match the closest training feature. The number of test features that are found to match a training feature and the number of times each training feature is found to match a test feature are recorded. If a training feature is found to match more than one test feature, all matches found using this feature are removed from the number of total matches; this provides a way to easily remove incorrect matches resulting from a single training feature being matched repeatedly to a number of test SIFT features. While previous applications have used more sophisticated methods to determine SIFT matches (including geometrical constraints on matched features [73], [122]), this simple matching procedure has been sufficient for object matching in the CRS.

In the CRS, SIFT features have been used to recognize a book and a doll. Figures 4.9 and 4.10 show the locations of matched SIFT features between training and test images. While the figures show erroneous matches (red lines that end on locations in the images that do not correspond to the book or doll), the majority of the lines connect the same location on the object in the training and test images, indicating a useful match.



Figure 4.9: SIFT matches between a training image (left) and test image (right) used to recognize a book.

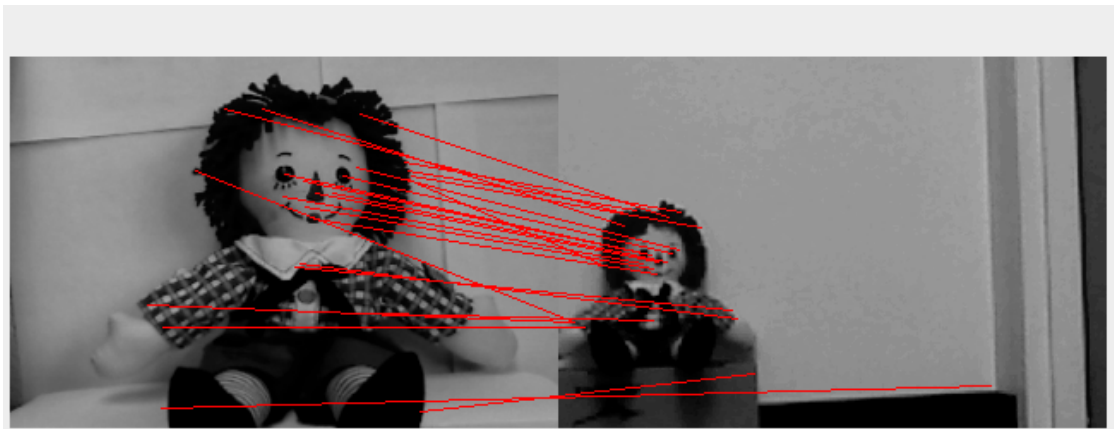


Figure 4.10: SIFT matches between a training image (left) and test image (right) used to recognize a doll.

Outdoor navigation

This chapter describes the system used for an outdoor navigation mission (Section 2.5.1) and the results from tests using two different Soar agents are described.

5.1 CRS description for outdoor navigation

The outdoor navigation mission was performed using the SuperDroid (Section 3.1) as the robotic platform. Two sonar distance sensors, described in Section 3.3.1, were used to detect obstacles in front of the SuperDroid. One sensor was used to sense the left half of the area in front of the robot and the other sensor sensed the right half of the area in front of the robot. Two infrared distance sensors (the GP2D12 model, with the shorter range from Section 3.3.2) were placed to sense obstacles to the left and right sides of the robot. The electronic compass described in Section 3.3.3 and the GPS unit from Section 3.3.4 were also used in this system. Figure 5.1 shows the connections between a BrainStem module (Section 3.2.2), a laptop (Section 3.2.1), the MD22 motor driver (Section 3.2.3), and the sensors and motors used for this mission.

The BrainStem communicates with an electronic compass, two sonar distance sensors, and two infrared distance sensors and sends the sensor information to the onboard PC and sends motor output commands to the motor driver board. The laptop was used to run the Java program and Soar agent used by the CRS. Figure 5.2 shows the integration of the software components used by the CRS for the outdoor navigation mission. Starting at the bottom left of the figure, the

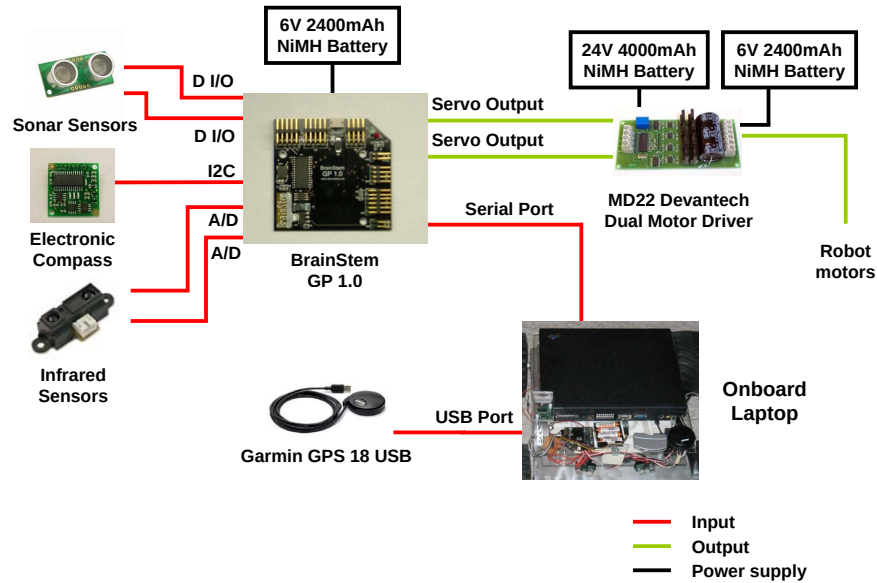


Figure 5.1: A schematic of the hardware used onboard the SuperDroid and their connections. Inputs to the BrainStem module and laptop are shown as red lines. Output from the laptop and Devantech Motor Driver are shown by the green lines. The black lines represent power supplied to the BrainStem module and the Devantech Motor Driver.

Java program retrieves sensor information (a compass measurement and range measurements from the sonar and infrared distance sensors) from the BrainStem module and receives and processes information from the GPS unit. The Java program then uses the Soar Markup Language (SML) to place the relevant sensor information on the Soar agent's input-link and then to execute the Soar agent for one decision cycle (represented by the input box, the dashed box representing the propose, decision, and apply phases and containing a hierarchy of operators for the outdoor navigation agent, and the output box), and finally to check the output-link for new commands. If there is not a command on the output-link, the Java program executes the Soar agent for another decision cycle. If there is a motor command available, the Java program sends that command to the BrainStem module before beginning the cycle again. More details about the Java program are available in

Janrathitikarn [126] and Hanford, Janrathitikarn, and Long [16].

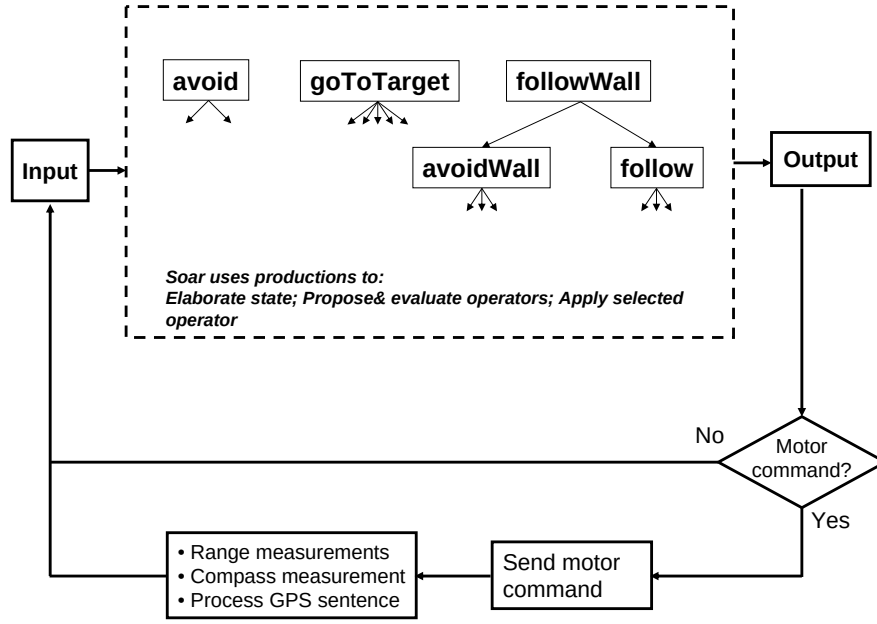


Figure 5.2: Integration of the Soar architecture and other software in the CRS.

5.2 Description of Soar agents

Two Soar agents (called the original and updated agents) were developed for the mission of navigating to a GPS location while avoiding obstacles. The agents are similar, but the updated agent uses some additional sensor information and operators to extend the capabilities of the original agent.

5.2.1 Original Soar agent

The original Soar agent uses 57 production rules, including 2 complex operators and 10 primitive operators, to make decisions and generate motor commands. The Soar agent receives inputs from Java: the distances to the closest obstacles sensed by the two sonar sensors, the robot's current heading, the difference in latitude

and longitude between the current GPS coordinate and the goal location, and the current clock value (this value is increased by one for each time new input is sent to the Soar agent). The only possible output from each Soar agent is a motor command. The working memory for the agent includes information such as:

- if the obstacles sensed by the range sensors are close enough to the robot that the agent needs to consider avoiding them
- the current distance and heading to the goal location
- the difference between the robot's heading and the heading to the target
- information such as the actions of recent operators and the times they were applied

This agent used two complex (or abstract) operators: `avoid` and `goToTarget`. The `avoid` operator is proposed when at least one sonar sensor detects an obstacle and the `goToTarget` operator is proposed when no obstacles are detected by the sonar sensors. When the `avoid` or `goToTarget` abstract operators are selected, the Soar agent implements them in subgoals, where a new state is created and named `avoid` or `goToTarget`, respectively. In these subgoals, relevant low-level operators can be proposed, selected, and applied to accomplish the goal of the selected abstract operator. The hierarchy of operators for this Soar agent is shown in Figure 5.3.

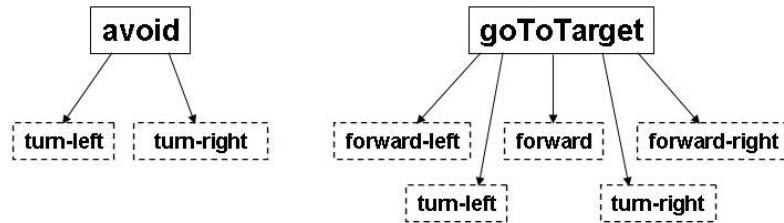


Figure 5.3: The hierarchical structure of the `avoid` and `goToTarget` abstract operators and their low-level, or primitive, operators of the original Soar agent is shown. The abstract operators are shown in the boxes with solid borders and the low-level operators are shown in boxes with dashed borders.

In the avoid subgoal, the turn-left and turn-right primitive operators are eligible to be proposed and are used to avoid obstacles sensed by the sonar sensors. The operators are used to turn away from the obstacles (e.g. if the agent senses an obstacle on its left, the turn-right operator is proposed). If the agent senses obstacles to both the left and right, both the turn-left and turn-right operators are proposed and one is chosen at random. In the goToTarget subgoal, the turn-left, forward-left, forward, forward-right, and turn-right operators are eligible to be proposed. The proposal of these five operators depends on the difference between the heading required to go to the target location and the current heading of the robot. If this difference in heading is small, the forward operator will be proposed. If the difference in heading is larger, the forward-left, forward-right, turn-left, and turn-right operators are proposed. Additionally, the forward operator is always proposed in case all other proposed operators will have undesirable effects. The proposal of operators in the avoid and goToTarget subgoals are mostly reactive: they mainly consider the current state of the agent and do not consider the agent's past. However, the Soar agent also remembers a history of the last ten operators selected during its execution that can be used to create preferences for operator selection when more than one operator has been proposed. These preferences ensure that a proposed operator that will undo the actions of recent operators will not be selected for application unless it is the only proposed operator. Recording the last ten operators can be useful when avoiding obstacles. For example, without using the memory of recent operators, the agent could turn left to go to a target location, see an obstacle to its left, turn right to avoid the obstacle, then turn left to the target, and immediately see the same obstacle again.

5.2.2 Updated Soar agent

The updated Soar agent uses additional sensor information, remembers more about its recent actions, and is able to use more than one strategy to avoid complicated obstacles. In addition to the sensors used by the original Soar agent, the updated Soar agent used two infrared distance sensors to measure the distances to the closest obstacles on the right and left sides of the robots. This agent has 104 production rules, including 6 complex operators and 21 primitive operators. The

updated Soar agent also remembers more about the operators it has previously applied than the original agent, letting the Soar agent consider in more detail what it has done recently. This agent records the names of the last 25 operators (forward, turn-right, etc.) it has applied, what subgoal they were applied in (goToTarget, avoid, etc.), and what actions (turn command, throttle command) were associated with the operators. In addition to using this information to avoid undoing recent actions, the Soar agent uses the information to recognize when the robot is sensing obstacles repeatedly on the same side of the robot. The updated Soar agent uses three abstract operators at the top-level of the hierarchy (the followWall abstract operator also has abstract operators) to move to a desired location: avoid, goToTarget, and followWall (Figure 5.4). The avoid abstract operator is proposed when at least one sonar or infrared sensor detects an obstacle and the robot is facing towards the target (it has a heading within 90 degrees of the desired heading). The goToTarget operator is proposed when no obstacles are detected by the sonar and infrared distance sensors. The followWall abstract operator is proposed for two different situations: 1) when the robot is repeatedly encountering an obstacle on one of its sides or 2) when the robot senses an obstacle while facing away from the target (it has a heading greater than 90 degrees away from the desired heading). The followWall operator has a higher priority than the goToTarget operator and is preferred for selection if both operators are proposed at the same time.

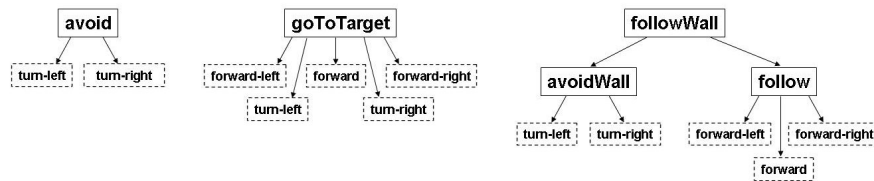


Figure 5.4: The hierarchical structure of the operators and their low-level operators is shown. The abstract operators are shown in the boxes with solid borders and the low-level operators are shown in boxes with dashed borders.

The goToTarget and avoid subgoals for the updated Soar agent are similar to those for the original Soar agent. The avoid subgoal, however, uses the additional information from the infrared distance sensors and detailed memories of the agent's recently applied operators while selecting an operator to apply.

Once the followWall operator has been selected and the Soar agent is in the

followWall subgoal, the agent must decide if it should follow a wall on its right or left. This decision is made by considering which side of the robot the sensed obstacle is on that resulted in the selection of the followWall operator. In the followWall subgoal, the Soar agent attempts to keep the robot a set distance from the closest obstacle sensed by the infrared distance sensor on the side of the robot on which the wall is being followed.

The Soar agent uses two abstract operators to accomplish its goal of following a wall. The first is the avoidWall operator, which is proposed if either of the sonar sensors sense an obstacle. In this avoidWall subgoal, the turn-right and turn-left operators are used to turn away from the side of the robot on which the wall is being followed. For example, if the robot is following a wall on its right side and senses an obstacle using either of its sonar sensors, the Soar agent instructs the robot to turn left, away from the wall it is following. If no obstacles are detected by the sonar sensors, the follow abstract operator is proposed. In the follow subgoal, the infrared distance sensor on the side of the wall being followed (left side sensor for left wall following, right side sensor for right wall following) is used to determine the distance between the robot and the closest obstacle and then how the robot should use the forward, forward-left, and forward-right operators to maintain the desired distance from the wall. The goal of following a wall is considered to be accomplished (and the agent therefore exits the followWall subgoal) when the robot's heading is within 30 degrees of the heading to the target and no obstacles are sensed by any of the infrared or sonar sensors.

5.3 Tests and results

Experiments were performed with different types of obstacles to test whether the Cognitive Robotic System could control the robot to navigate to a GPS location while avoiding obstacles. These tests were performed at the top level of a parking garage at the Pennsylvania State University. A picture of the testing environment is shown in Figure 5.5 and a schematic of the parking lot is depicted along with the results in Figures 5.6, 5.7, and 5.8. The thick black line in these three figures represents the walls on this level of the parking garage. In addition to the walls, there is a stairway in the bottom right of the figures that provided another obstacle

to the robot. The lower half, the far left of the upper half, and the far right of the upper half of the figures (dark yellow area) are level. The upper-left part of the figures is a ramp up to the additional parking spaces. The upper-right part of the figures is a ramp down to a lower level of the parking garage. The red 'X' in the lower right corner indicates the target GPS location. For these tests, the robot was considered to have reached the target when the robot was within 3 meters of the specified GPS location.



Figure 5.5: A picture of the top level of the parking garage on which the tests described in this paper were performed. This picture was taken from a location in the lower left corner of Figures 5.6, 5.7, and 5.8.

5.3.1 Results for original Soar agent

The original Soar agent was successfully able to complete the mission for simple obstacles that could be avoided without using sophisticated reasoning. For example, in Figure 5.6, eleven obstacles were placed between the robot's starting position in the lower left corner of the figure and the target in the lower right corner of the figure. The obstacles were placed far enough apart so that the robot could move between any of the obstacles. The results shown in Figure 5.6 were obtained using an implementation of the CRS on a hexapod, or six-legged, robot (described in depth by Janrathitikarn [126] and Hanford, Janrathitikarn, and Long [16]). The same original Soar agent was used for this test on the hexapod and the test shown in Figure 5.7, which was performed using the SuperDroid.

However, the original Soar agent was not able to complete the navigation mission in the presence of a complex cul-de-sac obstacle, as shown by the results in Figure 5.7. The robot started in the upper left corner and moved to the target using the `goToTarget` operator until it sensed an obstacle (the wall in the middle

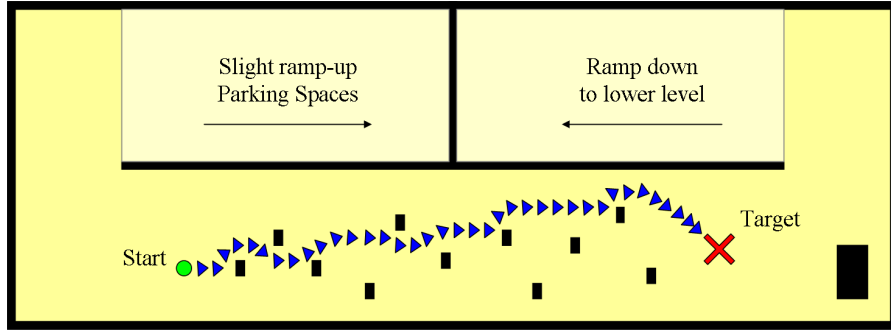


Figure 5.6: The results for a test with simple obstacles are shown. The hexapod started in the lower left corner of the figure at the green circle and was able to successfully avoid the obstacles and reach the target location by switching between the avoid and goToTarget problem spaces.

of the upper part of Figure 5.7). The robot then turned right using the avoid operator. When it no longer detected the obstacle, it returned to using the goToTarget operator. The target was then behind and to the right of the robot, so the robot turned to the right. For clarity, the path of the robot for the rest of the test is not shown, but the robot attempted to move to the target until it encountered the wall in the middle of the upper part of Figure 5.7 again. The robot then again used the avoid operator to move itself out of the corner. The robot repeated this pattern of behavior, continuing to move into and out of the corner of the cul-de-sac in the upper left corner of Figure 5.7, until the test was terminated unsuccessfully. These tests demonstrated that the original Soar agent was capable of guiding a robot to a GPS target location with relatively simple obstacles. However, when the agent was not able to reactively avoid obstacles, it could not successfully navigate to the target location.

5.3.2 Results for updated Soar agent

This section describes the results from a test in which the CRS implementation with two infrared sensors and the updated Soar agent was used to repeat the test with the cul-de-sac obstacle shown in Figure 5.7. The SuperDroid was placed at the location in Figure 5.8 marked at the green circle labeled 'start' for this test. The Soar agent used the goToTarget subgoal to turn to the right and start moving toward the target.

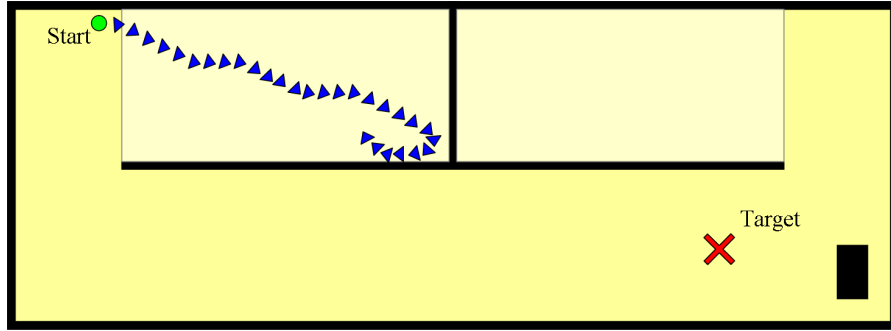


Figure 5.7: The results of a test with a cul-de-sac obstacle are shown. The SuperDroid started in the upper left corner of the figure at the green circle and was not able to successfully reach the target location because it became stuck in the cul-de-sac in the upper left corner of the figure.

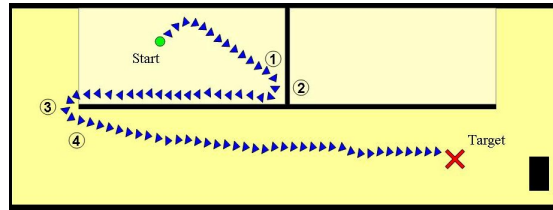


Figure 5.8: The results of the test of the SuperDroid, in which the updated Soar agent navigates to the target location in the presence of a cul-de-sac, are shown.

At the point labeled 1 in Figure 5.8, the robot sensed the wall to its left as an obstacle and used the avoid subgoal to turn to the right until it had moved far enough so that the wall was not considered an obstacle. At this point (when no obstacles are detected), the goToTarget abstract operator was selected. In the goToTarget subgoal, the forward-left and forward operators are proposed: the forward-left operator since the target is to the left of the robot and the forward operator since it is always proposed in the goToTarget subgoal in case the only other option is to undo a recently performed action. Since the robot just turned right to avoid an obstacle, the forward operator was selected instead of the forward-left operator to avoid undoing the actions of the recently applied turn-right operator. The Soar agent selected the forward operator for a total of five decision cycles, choosing to select this operator instead of the forward-left operators that were also proposed since the last-avoid-operator (the turn-right operator used in the avoid subgoal) was remembered for five cycles. After five cycles, the last-avoid-operator was forgotten and the forward-left operator was chosen. Within two decision cy-

cles, the robot was again close enough to the wall for the wall to be considered an obstacle, resulting in the use of the avoid subgoal to move away from the wall again using the turn-right operator.

The process described in the preceding paragraph repeated itself until point 2 in Figure 5.8, when the agent decided that the SuperDroid was repeatedly encountering and having trouble avoiding an obstacle on its left and proposed and selected the followWall abstract operator. In the followWall subgoal, the agent then chose to follow the wall on its left side. The agent was able to turn the corner near point 2 (in Figure 5.8) using the avoidWall subgoal after the horizontal wall in Figure 5.8 had been sensed by the robot’s sonar sensors.

The Soar agent remained in the followWall subgoal while it followed the wall on the left side of the robot and continued to follow the wall around the 180 degree turn in the wall near point 3 in Figure 5.8.

Once the robot reached point 4 in Figure 5.8, the Soar agent exited the followWall subgoal since its heading was within 30 degrees of the heading to the target and no obstacles were sensed. The agent then entered the goToTarget subgoal and used the low-level operators for this subgoal to successfully navigate to the target location.

5.4 Discussion

This chapter described how the CRS was used to integrate Soar with the sensors and motors of an unmanned ground vehicle. Two Soar agents were developed to test the CRS on a practical mission of navigation with obstacle avoidance. The original, simpler agent was able to successfully complete a test in a relatively simple environment, but was unable to navigate in the presence of a cul-de-sac obstacle. The updated agent was able to use two different approaches to avoiding obstacles to successfully complete the test with a cul-de-sac obstacle without the use of a map. The idea of using a simple approach, but realizing when the simple approach is not working as desired and switching to a more complicated (potentially more computationally expensive) approach, could be useful for robot applications other than the navigation mission described in this chapter. The success of this agent also demonstrated the potential for using additional sensor information and

more Soar rules for creating more capable Soar agents.

There are many examples of robots that are capable of navigating out of a cul-de-sac or sensing an cul-de-sac before entering it at all. Unlike the updated Soar agent described here, these systems generally use a map or long-range perception capabilities (as was common for systems used in the DARPA LAGR program [9], [10]). As mentioned in Section 2.5.1, a version of the software for the Mars Exploration Rovers was not able to navigate around a collection of rocks to a goal location behind the rocks. The rover software was updated after this weakness was discovered to use a version of the Field D* algorithm for global path planning using a 2D grid-based map of the environment, with the goal of allowing the rover to autonomously solve similar problems [74]. Yamauchi [111] described a robot with an obstacle avoidance system that could be used to navigate out of a cul-de-sac. This system considers both obstacles identified in an occupancy grid and the robot's recent locations as obstacles that should be avoided.

There are changes to the CRS that would be useful or interesting to explore for the mission described in this chapter. First, developing a Soar agent that is able to use Soar's episodic memory or reinforcement learning mechanisms could have potential for making decisions to avoid entering or navigating out of a cul-de-sac. Incorporating the ability for the CRS to generate a map of its environment would be another useful change. There is not a guarantee that the updated Soar agent's approach to changing the operator it uses to avoid obstacles will always be successful for navigating out of a cul-de-sac. If the CRS generated a map, the map could be used as another source of information for avoiding entering or navigating out of a cul-de-sac. There are several ways in which the CRS could utilize the information stored in a map.

- Currently, if the robot revisits a cul-de-sac, the CRS would have had no memory that there was a cul-de-sac and would have to repeat the process of first having difficulty avoiding the obstacle using its simpler avoid subgoal and then choosing its wallFollow subgoal to get out of the cul-de-sac. With a map, the Soar agent could have remembered there was an obstacle to avoid in that area and planned a different path to reach its target location. (Similar behavior might also be achievable using Soar's episodic memory or reinforcement learning).

- The Soar agent had a difficult time knowing when it had moved past an obstacle and could resume going towards its goal location. The agent attempted to solve this problem by waiting a specified number of decisions before applying an operator that would undo the action the agent took to avoid an obstacle. For example, if the robot turned right to avoid an obstacle, the agent would try to avoid turning back to its left for a specific number of decisions. If the agent had access to a map, it could have used the locations of the obstacle and robot on the map to reason about if the robot was past the obstacle and it was worthwhile for the robot to try to turn left to go to the target location.
- The use of the CRS to perform the navigation mission discussed in this chapter does not preclude the use of specialized path planning approaches. For example, the Soar agent could try using its current production rules to accomplish the navigation mission, but realize if it is having difficulty completing the mission that it could decide to use a map and a path planning algorithm to calculate a path the robot could use to reach the target location.

The outdoor navigation mission described in this chapter could be accomplished with minimal amount of high-level information and without a formal internal model of the environment. However, in general, incorporating improved perceptual systems (e.g. computer vision object recognition algorithms) or traditional robot capabilities like map building that could be used to find meaningful symbols from the environment for a Soar agent to use in its reasoning could substantially increase the capabilities of the CRS.

Indoor search

The first three sections of this chapter describe components of the CRS used for the indoor search mission. Section 6.1 describes the hardware components (from Chapter 3), Section 6.2 describes the software and algorithms (from Chapter 4), and Section 6.3 describes the Soar agent used for this mission. Some details about how the CRS uses the components described in these three sections and the results of several tests (performed both using the SuperDroid and in simulation) are given in Section 6.4. Section 6.5 discusses possible improvements to the system described in this chapter, as well as providing comparisons to systems developed by other researchers.

6.1 Hardware for indoor search mission

The SuperDroid (Section 3.1) was used as the robotic platform for the system developed for the indoor search mission. The robot used three sonar distance sensors (described in Section 3.3.1): one facing forward, one facing to the left, and one facing to the right. Two infrared distance sensors (the GP2Y0A02YK0F model, with the longer range from Section 3.3.2) were placed to face the left and right sides of the robot. A wheel encoder (Section 3.3.5) was installed on each of the front wheels of the SuperDroid. One webcam, described in Section 3.3.6, was used for the indoor search mission. Figure 6.1 shows the connections between the two BrainStem modules (Section 3.2.2), a laptop (Section 3.2.1), the MD22 motor driver (Section 3.2.3), and the other hardware used for this mission.

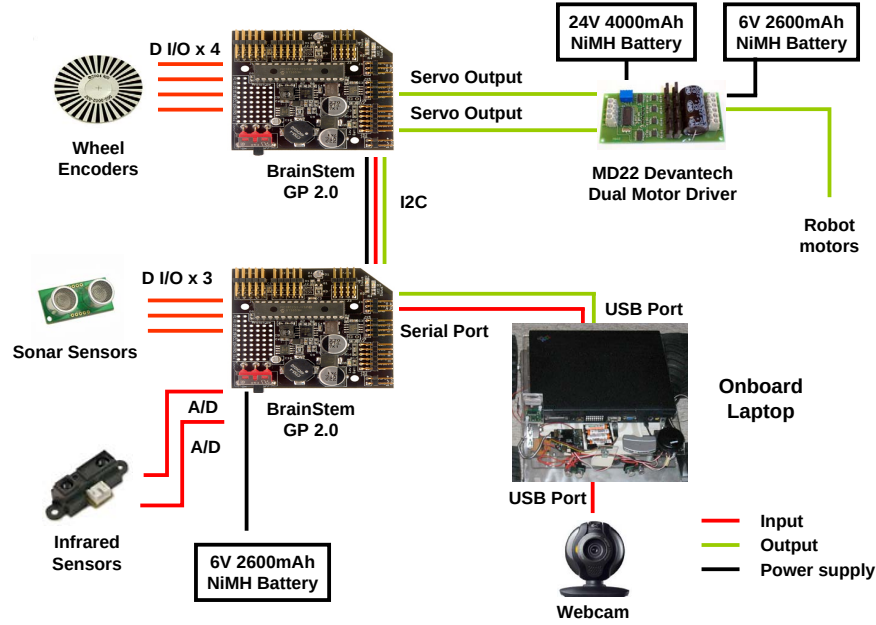


Figure 6.1: A schematic of the hardware used onboard the SuperDroid and their connections. Inputs to the BrainStem modules and laptop are shown as red lines. Outputs from the laptop, BrainStem modules, and Devantech Motor Driver are shown as green lines. The black lines represent power supplied to the BrainStem modules and the Devantech Motor Driver. The second BrainStem GP 2.0 module communicates with and receives power from the first BrainStem GP 2.0 module using its I2C ports. There is a USB to serial converter between the laptop and first BrainStem module.

One BrainStem is used to communicate with the three sonar and two infrared range sensors. The second BrainStem communicates with the encoders on the front wheels of the SuperDroid and sends motor signals to the MD22 motor driver board, which is used to control the SuperDroid motors. The laptop sends and receives information from both BrainStem modules through a USB port and receives information from a web camera through another USB port. The second BrainStem module has a different module address from the first BrainStem and communicates with the laptop through an I2C connection with the first BrainStem. The first BrainStem acts as a router, sending information from the second

Table 6.1: Sensors and uses for sensor information in indoor search mission

Sensor	Uses for sensor data
Sonar	Occupancy grid, navigation
Infrared	Occupancy grid, navigation
Wheel encoders	State estimation, motor control
Webcam	Input to SIFT program

BrainStem to the laptop and information from host applications on the laptop to the second BrainStem.

Table 6.1 shows the sensors used for the indoor search mission and how the CRS uses the information from these sensors. The sonar and infrared range sensors are used as a source of information for the occupancy grid and by the Soar agent to make navigation decisions. Information from the wheel encoders is used for state estimation and motor control and images from the webcam are used by the SIFT program.

6.2 Software and algorithms

This section describes the software and algorithms used by the CRS for the indoor search mission, including both objects in Java executed on the laptop and TEA programs executed on the BrainStem module. Table 6.2 summarizes how the results of the algorithms described in Chapter 4 are used by the CRS. A Java program is used to integrate information from the sensors described in Section 6.1 and these algorithms and communicate with and execute the Soar agent described in Section 6.3 using the Soar Markup Language (SML). More details about how the CRS integrates Java, Soar, and programs on the BrainStem modules will be described in Section 6.4.

This list describes how the CRS uses its various processors to implement these algorithms:

State estimation: Equations that estimate the robot state (described in Section 4.1) are implemented by a Java object that uses odometry information generated by a Tiny Embedded Application (TEA) program on the BrainStem

Table 6.2: Algorithms and uses of results for indoor search mission

Algorithm	Uses for results
State estimation	Occupancy grid, Soar agent
Motor control	Drives robot motors
Occupancy grid	Hough transform, virtual rangefinder
Hough transform	Hall width and orientation
Virtual rangefinder	Navigation, inputs to fuzzy logic system
Fuzzy logic system	Intersection confidences
SIFT	Recognize object(s) of interest

module that monitors the wheel encoders to estimate the robot's position (described by x , y , θ coordinates). (GPS signals are not available in indoor environments.) The estimated state is used by the occupancy grid algorithm and the Soar agent.

Motor control: The PI motor control algorithm described in Section 4.2 is performed by a TEA program that is executed by the second BrainStem module, which sends the outputs of this program to the motor driver board.

Occupancy grid: Java objects are used to implement the occupancy grid algorithm and inverse sensor models for the sonar and infrared sensors described in Sections 4.3.2.1 and 4.3.2.2. The estimated robot state and measurements from the distance sensors are used to generate a local occupancy grid (4m by 4m) that is centered on and moves with the robot. The occupancy grid provides a framework to fuse information about obstacles from all five distance sensors and is used by the Hough transform and virtual rangefinder algorithms.

Hough transform: The Hough transform is performed using a Java object and is used to identify the dominant lines in an image of the occupancy grid. As described in Section 4.4, the results of this algorithm are used to estimate the current hall's width and orientation, information which is used by the fuzzy logic system for identifying intersections and by the Soar agent.

Virtual rangefinder: The virtual rangefinder described in Section 4.5.2.1 was

also implemented using a Java object. The results of this algorithm are used by both the fuzzy logic system for intersection detection and the Soar agent for navigation.

Fuzzy logic system: The fuzzy logic algorithm (Section 4.5.2.2) used for generating confidence values for the presence of types of intersections is implemented in a Java object and the results are used by the Soar agent for detecting intersections.

SIFT: A Java object is used to communicate with the SIFT executable described in Section 4.6.3 and process the results of this executable to recognize objects of interest.

6.3 Description of Soar agent

The Soar agent described in this chapter is capable of performing several tasks:

- moving through a building using two types of navigation: low-level corridor following and moving to specified waypoints
- recording intersections and making decisions about how to move through its environment based on these intersections
- building a topological map that can be used (along with Soar’s planning capability) to make intelligent decisions to efficiently return to beginning of test after finding the objects of interest
- detecting objects of interest and describing their locations relative to intersections on a topological map
- generating verbal instructions from the starting location of the test to an object of interest that can be shared with another robot or a person.

To perform these tasks, the Soar agent uses 729 productions (including 72 rules downloaded with Soar to use Soar’s planning mechanism). These productions implement 47 complex operators and 121 primitive operators.

6.3.1 Top-level operators

Figure 6.2 shows the top level of the operator hierarchy for this agent. A complete list of the operator hierarchy for the agent is given in Appendix B. The initialize-hallNavigator operator sets up some of the structure in working memory and initializes the Soar agent with information about its search task, such as how many and what types of objects of interest the agent must search for in the current test. If the occupancy grid is initialized or reset, the robot must travel a minimum distance before accumulating enough information in the occupancy grid to accurately reflect the location of walls in its environment. The initialize-intersectionDetectionAndHoughInfo operator is used to enable recording Hough information and the detection of intersections once the occupancy grid has enough information for the Hough transform to accurately identify the walls of the current hall.



Figure 6.2: The top level operators for the indoor search agent are shown. Abstract operators are shown in the boxes with solid borders and the primitive operators are shown in boxes with dashed borders.

The search operator is used to navigate through the building, detecting intersections and objects of interest, and generating a topological map until all of the desired objects are found. The details of this abstract operator are described in Section 6.3.4.

The returnHome operator is proposed once all the objects of interest are located. In this operator, the agent uses information from the topological map to return to the location where the search started. The details of this abstract operator are described in Section 6.3.5.

The record-intersection operator is used to record the type of intersection the robot is in and other information such as the time and location the intersection was first detected. Because the Soar agent has incomplete and sometimes incorrect information about its environment, it is important to have limits on when intersec-

tions can be recorded to limit incorrect detections of intersections. To attempt to minimize incorrect detections, the agent must note the presence of an intersection over time before it is recorded. While more specific details of when the agent can note the presence of an intersection will be given later, in general the occupancy grid must have contain enough information about the environment to accurately estimate the location of any walls represented in the map and the robot must be in a position in which its sensors can accurately represent intersections on an occupancy grid (e.g. aligned with and near the middle of a hallway).

The record-object operator is proposed when the number of SIFT matches between the current camera image and an image of an object of interest exceeds a threshold. The type of object and the time and location of the robot when the object is detected are recorded in working memory. The information recorded about the object can be used to describe the object's location relative to the intersections that have been recorded.

The search and returnHome operators can not be proposed at the same time: the search operator is only proposed before all objects of interest are detected and the returnHome operator is proposed after all objects of interest have been detected. However, the record-object, record-intersection, initialize-intersection-DetectionAndHoughInfo operators can be proposed at the same time as the search and returnHome operators. The record-object, record-intersection, initialize-intersectionDetectionAndHoughInfo operators are proposed with best preferences and are therefore selected for application before the search and returnHome operators. If the conditions that caused the search and returnHome operators to be proposed are still true after the application of the record-object, record-intersection, initialize-intersectionDetectionAndHoughInfo operators, the search and returnHome operators will remain proposed and can be selected for application.

6.3.2 Input to and output from agent

This section lists the information that a Java program sends to and retrieves from the Soar agent using the Soar Markup Language (SML). The following information is generated from sensors and sensor processing algorithms and placed on the Soar agent's input link:

- The distance to the closest obstacle sensed by each of the three sonar and two infrared distance sensors
- The estimated x and y position and orientation of the robot
- The orientation and number of contributors for both the left and right Hough lines and the distance between the two lines
- The confidence and sorted rank value for each of the eight intersection types.
- The average distances in the wedges to the left and right of the virtual rangefinder
- The number of current SIFT matches for each object of interest
- The current clock value (this value is increased by one for each time new input is sent to the Soar agent).

The Soar agent can place several types of commands on its output link for the Java program to process and execute, such as:

- Motor commands that include parameters such as the steer and throttle settings and the amount of time the motors are to be on
- A command to reset the values of all of the cells in the occupancy grid to their initial value
- When the Soar agent detects an object of interest, it places a command on its output link to save the image used to detect the object with a filename that includes the name of the type of object detected and the time at which it was detected
- After the Soar agent has found all of the objects of interest, it issues a command for the CRS to stop looking for objects of interest (no more images are taken with the webcam and processed using SIFT)
- A command to update the hall width used by the fuzzy logic system to detect intersections can also be issued by the agent

- A set of commands that can be used to send the step by step instructions from the agent's starting location to a detected object of interest to the Java program.

6.3.3 Working memory

This section describes the information that the Soar agent stores in its working memory. The working memory elements that contain this information are used for a wide variety of purposes and can be described in groups that contain information about the following areas:

Robot state: Information in working memory about the robot includes its state (x , y , and θ), its current compass direction, the relative angle between the robot and its current hall, and if the robot is aligned with the hall (based on a previously defined threshold). Elaborations are used to determine if there is a wall to follow on each side of the robot.

Ranges: Measurements from the three sonar distance sensors are compared to threshold values to determine if there are obstacles around the robot that need to be avoided.

Hall information: If there are a large enough number of contributors for both Hough lines, the Hough information from the input-link is considered to potentially represent a hallway and the right and left line orientations are averaged to calculate an estimate of the orientation of that hallway on the occupancy grid. Then, if the orientation of this potential hallway makes sense given the robot's current orientation, this hall orientation and width are saved in working memory.

Intersection information: The two intersection types with the highest confidence values are identified. The confidence values of these types are compared to determine if the detection algorithm is confident enough in the type with the highest confidence to mark the the type with highest confidence as a "good detection" and consider noting the intersection. If the confidence of second highest type is greater than zero, the ratio of the highest confidence to the

second highest confidence is calculated. If this ratio is above a threshold, a good detection is noted for the type with the highest confidence. This test prevents a case where the two highest confidence values are very close (for example, 0.55 and 0.45) from being considered a good detection. If there is only one non-zero confidence value, the highest confidence value must be higher than a specified threshold. This test prevents a case with a low highest confidence value from being considered a good detection. The type of intersection and time sensed of any intersection detection that is marked as a good detection is saved in working memory for five time steps.

Recorded intersections and objects: The agent records information about the intersections and objects it detects including the time and location when the intersection or object was first and last detected, the type of intersection or object, how many intersections or objects have been previously detected, the compass direction of the robot during the detection, and if the intersection has been visited before.

Navigation plans: All the plans the agent creates to move through intersections are remembered. The reason the plan was created, the action of the plan, the series of goals needed to complete the plan, and information about the status of the plan (the time it was started, if the plan was successful, if the plan is currently active) are also recorded. Information recorded about each goal in a plan include the goal's status (for example, active, completed, or to be done in future), its action, and its goal state (defined by x , y , and θ values). An active goal also has information such as the distance and angle to the goal waypoint that are used to determine if the agent has reached the desired waypoint and can change the goal's status from active to complete.

Topological map: The agent's topological map stores information about nodes (intersections) and how nodes are connected to each other as well as objects that the agent has detected. Each node in the topological map has a unique ID number and stores information such as the types of intersection that comprise the node (a node can be seen as a different intersection type depending on the direction the node is entered from), information about connections to other known nodes (the node ID of adjacent nodes, the direction to these

nodes, and the length of the connections), and knowledge about when the robot has previously been in the node.

Information for generating instructions to object of interest: There are also some working memory elements created at the end of the test that are used to generate directions to the object of interest. These elements include information such as the chronological step number, the type of action to take, and the type of intersection (if there is one) in which the action should be performed.

The working memory elements describe above include both i-supported and o-supported elements. The Soar agent uses a set of i-supported elaborations to generate information about its current state that is only relevant for as long as the working memory elements used to match the conditions and fire the rule that creates the element are true. Examples of working elements that can be created with i-support are elaborations that describe if the front sonar sensor currently detects an obstacle, calculate how far the robot is from the goal state of an active goal, and whether there is currently a good intersection detection. Because these elements are i-supported, they are removed from working memory when the working memory elements used to create them are removed. For example, the distance to the location of an active goal is calculated with the x and y values of the robot's current location and the x and y values of the goal state. Once the robot's x and y locations are updated or the goal is no longer active, the working memory element containing the distance to the goal location is automatically removed. (While this can be very useful as a truth maintenance system, there are times where it would be useful to remember information such as this distance: for example, to compare the previous and current distances to the goal location to determine if the robot was moving closer to the goal).

However, there are also numerous working memory elements described above that must be remembered after the conditions that caused their creation are no longer true. These elements, which include the topological map, navigation plans, and information about the intersections and objects that have been recorded by the agent, must be created with o-supported rules. The rules that create these working memory elements will be explained in depth in the following sections.

6.3.4 Search

The search operator has three complex operators (Figure 6.3) that are used to explore a building using two types of navigation strategies. The navigate operator uses reactive navigation strategies such as following a wall or corridor while the perform-plan and set-plan operators support navigation between waypoints.

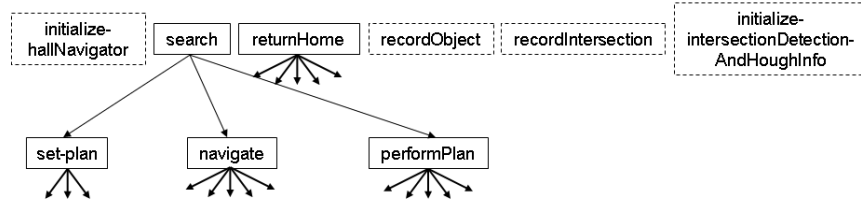


Figure 6.3: The three abstract operators for the search operator of the indoor search agent are shown.

6.3.4.1 Navigate

The goal of the navigate operator is to move the robot through a building while avoiding obstacles and keeping the robot in a position to accurately detect intersections. The navigate operator is proposed if the agent detects it is in a hall (‘S’ intersection) or if no intersection is currently detected. This complex operator uses four additional complex operators (follow middle, follow wall, follow orientation, and avoid obstacle) to achieve the goal of keeping the robot in a good position to detect intersections as it moves through a building; because of the limited sensors available on the robot, intersections are most easily detected if the robot stays near the middle of hallways with an orientation aligned with the hall.

The proposals of the operators in navigate are dependent on the presence or absence of walls to follow on the left and right sides of the robot. If any of the following three conditions are present, the agent assumes that there is not a wall to follow on the corresponding side of the robot:

- the infrared sensor on that side does not sense an obstacle in its field of view
- the virtual rangefinder corresponding to the region to the side of the robot does not sense an obstacle

- the side sonar sensor measurement is large compared to the current hall width.

Assuming there are no close obstacles in front of the robot, the Soar agent proposes the follow middle operator if there are walls to follow on both its left and right sides, the follow wall operator if there is a wall to follow on only one side, and the maintain orientation operator if there is not a wall on either side of the robot. If there is a close obstacle in front of the robot, the avoid obstacle operator is proposed.

If there is not an obstacle in front of the robot, a go-straight operator is proposed with a worst preference as a last resort to keep the robot moving if the conditions to propose the follow middle, follow wall, or maintain orientation operators are not present.

Follow middle: The follow middle operator is proposed when the Soar agent determines that there are walls to follow on both sides of the robot. In the follow middle subgoal, the agent can propose primitive operators (that can send motor commands to Java) based on the robot's location with respect to the middle of the hall and its relative orientation to the hall (the operators available for proposal in the follow middle subgoal are shown in Figure 6.4). The agent uses the difference between the two side sonar sensors to estimate the distance the robot is from the middle of the hall. Using this distance and a threshold defined working memory, the agent determines if the robot is in the middle of the hall or too far to the right or the left of the middle of the hall. If accurate information about the current hall's orientation is available (from Hough transform information), the agent also considers whether the robot is aligned with the current hall or if its orientation is to the left or right of the hall.

As an example of how the agent uses the robot's location in and alignment to the hall to propose operators, the forward-left operator is proposed in three situations: if the robot is to the right of the middle of the hall, if the robot is facing to the right of the hall's orientation and is located in the middle or to the right of the middle of the hall, or if the robot is aligned with the hall and is located to the right of the middle of the hall. Selecting an operator that was proposed using information about the alignment of the robot with respect to the hall is preferred

over selecting an operator that has been proposed using only information about the location of the robot with respect to the middle of the hall. The operators that are proposed by rules that do not have a condition that checks the alignment of the robot with respect to the hall are useful when the Hough information is not available or the potential walls identified from the Hough transforms are not aligned with the robot.

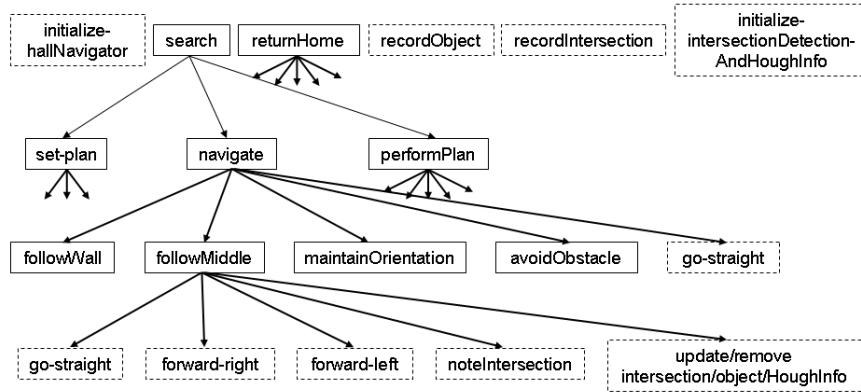


Figure 6.4: The operators used in the followMiddle abstract operator are shown.

As mentioned in Section 6.3.3, there are some working memory elements that need to be updated with o-support in support of the agent's goals. While operators that generate some of these elements fit well in the operator hierarchy for navigating through a building, it is less clear how operators that accomplish other goals (such as recording information about intersections and objects of interest) fit into this hierarchy.

In the Soar agent described in this chapter there are several o-supported working memory elements that need to be updated frequently by operators that don't directly contribute to the decomposition of the search or returnHome operators into primitive operators that can generate motor commands. An example of this is the case where the agent has selected the followMiddle operator and an object of interest has been detected. The agent can select the followMiddle operator only after generating a hierarchy of operators by selecting the search, navigate, and followMiddle operators. Once information indicating that the object of interest has been detected, the top-level operator recordObject is proposed. Since this op-

erator has a high priority (a preference to be selected over the search operator), it is selected for application over the previously selected (and still proposed) search operator. Since the search operator is no longer selected, all of the subgoals that were created to accomplish this goal (search, navigate, and followMiddle) are removed. The record-object operator is applied and if the search operator is still proposed after the application of the record-object operator, the search operator can be selected again, but the hierarchy of operators used to accomplish the search operator must be regenerated. To avoid the need for the Soar agent to regenerate its operator hierarchy every time one of these working memory elements must be updated, the agent includes operators that update these o-supported elements in all subgoals of the hierarchy that can propose primitive operators (instead of having these operators at the top level of the operator hierarchy). These operators accomplish tasks such as remembering any intersections detected in the last five time steps so the agent can differentiate between a single, potentially incorrect, intersection detection and repeatedly detecting the same type of intersection, updating the current hall width and orientation determined by the Hough transform, and updating information about intersections and objects of interest that have been recorded. The next paragraph discusses several operators used to perform these functions. Elaborations create operator preferences so that these operators will be chosen before operators that place primitive place motor commands on the output link. The next paragraph discusses several operators used to record and update o-supported working memory elements

The noteIntersection operator is able to create working memory elements that contain information about the type of intersection currently detected by the agent. In this followMiddle operator, the noteIntersection operator is proposed when the robot has moved far enough away from where the occupancy grid was initialized or reset, the potential walls identified by the Hough transform are aligned with the robot, the intersection type with the current highest confidence value is identified as a good detection, and the robot is located in the middle of the hall. While it is useful for the conditions for proposal of the noteIntersection operator to be different for different subgoals (and there are therefore different proposal rules in each subgoal that has primitive operators with motor commands), the remaining four operators are implemented by the same operator proposal and application rules

in all abstract operators where they can be used by the agent. The `removeIntersection` operator is used to remove information in working memory about intersections that were noted more than five time steps before the current time step. The `updateHoughInfo` stores the most recent estimation of the orientation and width of the robot's current hall. Once the intersection has been recorded by the `recordIntersection` operator, the `updateMapIntersection` operator is used to update the time and location of when the intersection was last detected. Similarly, once an object of interest has been recorded in working memory by the `recordObject` operator, information about when and where it was seen last is updated by the `updateMapObject` operator.

Follow wall: The purpose of the `followWall` operator is to stay in the middle of the hall in the absence of a wall to follow on one side of the robot. The agent attempts to use information from the sonar sensor on the side of the robot for which there is a wall to follow and the relative orientation of the robot to the orientation of the hall to stay aligned with the hall, a specified distance away from the wall it is following, and in a good position to detect intersections. The agent can use two sources of information to determine the distance the robot should be from the wall it is following to stay in the middle of the hall. If there is using information from the Hough transform available, the hall width is used to determine the desired distance from the wall to follow to the sonar distance sensor on that side of the robot. In case valid Hough information is not available, the sonar sensor reading on the same side of the robot as the wall the agent is trying to follow is recorded when the agent starts to use the `followWall` operator and used as the desired distance.

The `followWall` abstract operator has three primitive operators that can send motor commands to Java that help the Soar agent follow the desired wall. The side of the wall to follow (left or right), and the `distanceFromWall` and `headingRelativeToWall` attributes are used to propose the forward-left, forward-right, and go-straight operators. Elaborations are used to create the `distanceFromWall` attribute with a value of `too-far`, `too-close`, or `good` and the `headingRelativeToWall` attribute with a value of `aligned`, `facingWall`, or `awayFromWall`. For example, if the agent is trying to follow the wall to the robot's left and the value of the `distanceFromWall` attribute is `too-far` and the value of the `headingRelativeToWall`

attribute is aligned, the forward-left operator is proposed. Similar to the primitive operators with motor commands used in the followMiddle abstract operator, the operators that are proposed using both a condition that checks the angle of the robot relative to the hall (the headingRelativeToWall attribute) and a condition that checks the distanceFromWall attribute are preferred for selection over the operators that are proposed with only checking the distanceFromWall attribute.

The proposal conditions for the noteIntersection operator in the followWall subgoal are similar to the conditions for this operator in the followMiddle subgoal except that instead of requiring that the robot is in the middle of the hall as determined by the difference of the two side sensors, the robot is required to be close to the desired distance away from the wall it is following. The same removeIntersection, updateHoughInfo, updateMapIntersection, updateMapObject operators used in the followMiddle abstract operator and described for the followMiddle subgoal are used in the followWall abstract operator.

Maintain orientation: The purpose of the maintainOrientation operator is to stay aligned with the current hall if there are not walls to follow on either side of the robot. The agent uses elaborations to calculate the relative orientation between the robot and the hall, which is used to propose the three primitive operators (forward-left, forward-right, and go-straight) that are used to keep the robot aligned with the robot’s current hall. The maintainOrientation operator is useful when the robot is entering a ‘T’ or ‘X’ intersection.

The noteIntersection operator is proposed if the robot has moved away from where the occupancy grid was initialized or reset, the potential walls identified by the Hough transform are aligned with the robot, and the intersection type with the current highest confidence value is identified as a good detection. Again, the same removeIntersection, updateHoughInfo, updateMapIntersection, updateMapObject operators used in the followMiddle subgoal are used for the maintainOrientation abstract operator.

Avoid obstacle: Since the robot has limited information about the area in front of it, when the front sonar sensor detects an obstacle, the agent’s strategy is to wait in place until the presence of the obstacle is reflected in the robot’s occu-

pancy grid, allowing an intersection to be detected and a plan to navigate through that intersection to be created. The `avoidObstacle` abstract operator uses a `wait` operator that is always proposed when the `avoidObstacle` operator is selected to keep the robot from moving closer to the detected obstacle. The conditions for proposal of the `noteIntersection` operator are easier to satisfy than the conditions for the `noteIntersection` operators in the `followMiddle`, `followWall`, and `maintainOrientation` subgoals. The `noteIntersection` operator is proposed as long as the robot has moved away from when the occupancy grid was initialized or reset and an intersection type has been identified as having the highest confidence value; it is not required that the intersection with the highest confidence value be identified as a good detection. These more easily satisfied conditions help ensure that a plan is set to move away from the obstacle in front of the robot, but increase the likelihood that a plan be generated because an incorrectly identified intersection is recorded.

6.3.4.2 Setting plans in intersections

The `set-plan` operator is proposed anytime a new intersection except for an ‘S’ intersection is recorded or when an ‘S’ intersection is recorded while an agent is already executing a navigation plan. The goal of this operator is to create a navigation plan that will move the robot through the intersection. These navigation plans consist of one or more waypoint goals, each of which is defined by a desired robot state. To achieve this goal, the Soar agent uses knowledge about what plans are useful in specific intersection types and considers if the agent has been to the current intersection before to set a plan to move the robot through the newly recorded intersection. The `set-plan` operator also determines what to do if the agent records a new intersection as the agent is trying to perform a plan that has been set because of a previously recorded intersection. Although the agent uses the `navigate` operator to move the robot when an ‘S’ intersection is detected, recording an ‘S’ intersection can be useful for interrupting an active plan generated to move through an incorrectly detected intersection. In the `set-plan` subgoal, the agents uses elaborations to create elements in working memory that store information about the type of intersection the robot currently detects, the ID number of the current intersection, if there is already an active plan, and the ID number of

the last node visited in the topological map. These elaborations are used by the operators shown in Figure 6.5.

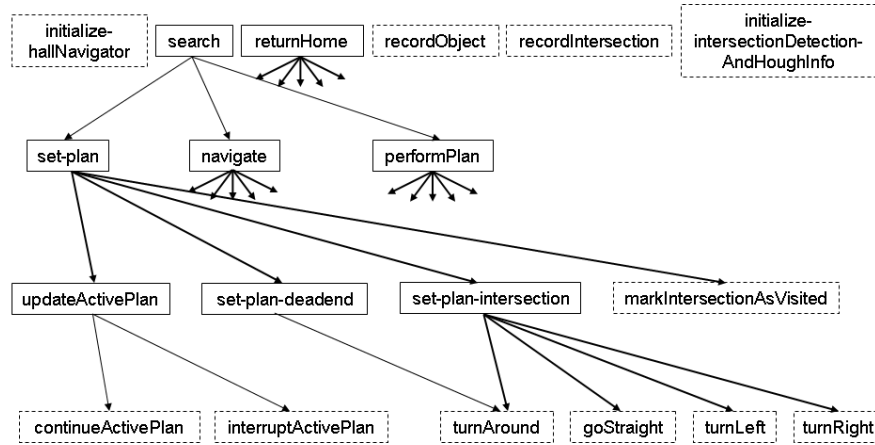


Figure 6.5: The operators used to generate a navigation plan are shown.

Determining if the agent has previously visited the intersection: As was mentioned in Section 4.3.1.2, it can be a difficult problem to identify if a robot has been visited an intersection before or if the intersection is being seen for the first time. To be able to deal with this problem, the current Soar agent assumes that the environment is static and without loops. These assumptions make it simpler for the Soar agent to use the information it has recorded about the current intersection and the topological map to recognize if it has been to the current intersection before and therefore propose the `markIntersectionAsVisited` operator. Additionally, there are Soar rules that create preferences that ensure the `markIntersectionAsVisited` operator is selected for application before the other operators proposed in the `set-plan` subgoal, allowing these other operators to make decisions with the knowledge that the robot is currently in a new intersection or one it has previously visited.

Reasoning about an active plan: The `updateActivePlan` operator is proposed when the Soar agent is trying to set a plan for a newly recorded intersection while there is another active plan. The goal of the `updateActivePlan` abstract operator

is to determine whether the agent should continue or stop performing the active plan.

There are two cases in which the agent proposes the `continueActivePlan` operator:

- if the newly recorded intersection is the same type as the intersection that is expected to be seen as a result of the plan the agent is currently performing (indicating that the robot may have already moved through the intersection)
- if the agent is currently performing a turning goal for the active plan (the occupancy grid is not a good representation of the robot's environment while the robot is performing turn goals).

There are two cases in which the agent proposes the `interruptActivePlan` operator:

- if the agent is performing the first go-straight goal of a plan and senses a new intersection that is not an expected result of performing the action that was chosen in the previous intersection
- if the agent is performing the first go-straight goal of a plan and the robot is still close to where the plan was started (as determined by a threshold defined to be 0.5 m).

If the `continueActivePlan` operator is selected, the agent resumes performing the active plan. If the `interruptActivePlan` operator is selected, the status of the current plan is changed from active to interrupted and a new plan can be set using the `set-plan-deadend` or `set-plan-intersection` operators.

Setting a plan in a dead end: If the agent is setting a plan for a 'D' intersection, the `set-plan-deadend` operator is used. The only possible operator is `turnAround`, which generates a plan with one goal: to turn around. After the robot has turned around, the `navigate` operator can be used to move away from the dead end.

Setting a plan in an intersection: If the type of intersection to set a plan for is a ‘L,’ ‘R,’ ‘T,’ ‘Tl,’ ‘Tr,’ or ‘X,’ the set-plan-intersection operator is used. All possible plans are proposed for the current intersection type (e.g. for an ‘R’ intersection, turning to the right and turning around are proposed). In intersections other than dead ends, the turn around plan is always proposed with a worst preference and only selected if a rule creates a preference stating that turning around is better than the other proposed operators. One plan is then selected based on each of the proposed operator’s preferences.

Navigation plans for turning to the left or right have three goals. The first waypoint goal is set to try to get the robot in the center of the intersection. This goal puts the robot in the center of the hall the robot is supposed to turn into and also allows the agent to confirm that the intersection that was recorded to trigger the navigation plan is correct (a ‘R’ intersection can sometimes be mistaken for a ‘Tl’ intersection if the front wall is too far away or a ‘R’ intersection can be mistaken for a ‘D’ intersection if the front wall is sensed before the opening to the right). The second goal in a turn right or turn left plan is to turn to the desired orientation and the third goal is to travel straight forward following the robot’s new orientation to get out of the intersection. Other plans have less than three goals. Plans for going straight through an intersection also only have one goal: moving forward. Plans for turning around in an intersection that is not a ‘D’ have two goals: a turn around goal and then a move forward goal to move out of the intersection.

Before the agent has detected all objects of interest (i.e. the agent is using the search operator), a plan is selected at random the first time the agent enters an intersection (except for the plan of turning around, which is selected only if there are no other options).

However, if the markIntersectionAsVisited operator has been applied, the agent uses information from the topological map about the connections to other nodes from the current node and the anticipated effects of the possible plans to help decide what plan to generate. Productions are used to compare possible plans and generate the following preferences:

- Prefer a plan that will lead to an unvisited area over a plan that leads to a visited area.

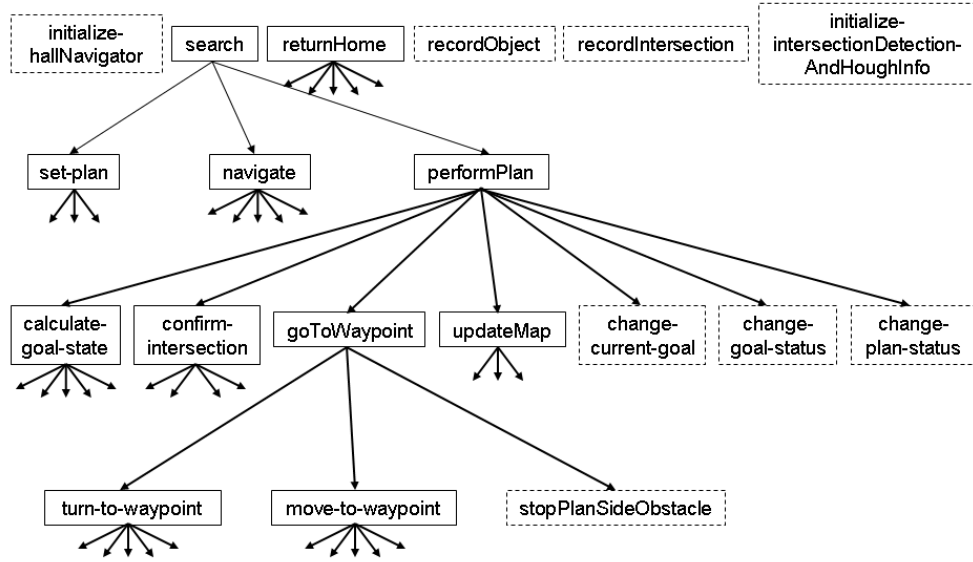


Figure 6.6: The operators used to perform a navigation plan are shown.

- Prefer a plan (including a turn around plan) that will lead to an already visited area over a plan that leads to the start location.
- Prefer a plan that will lead to an already visited intersection that is not a dead end over a plan that leads to an already visited intersection that is a dead end.

Operator preferences for setting specific plans could also be used to implement a more sophisticated search strategy.

6.3.4.3 Performing a navigation plan

Once a plan has been generated, the perform-plan operator is proposed to execute the plan. The perform-plan subgoal has operators that determine the current goal of the navigation plan and generate a goal waypoint for that goal, move the robot to the desired waypoint, and update the topological map after a navigation plan has been successfully completed. The hierarchy of the operators used to perform a plan are shown in Figure 6.6.

Determining the current goal waypoint: Several operators are used to update the status of plans and goals, to change the current goal in the navigation plan goal, and to calculate the desired waypoint state for the current goal. The change-current-goal operator is used to change a goal's status from future to active when the goal is the next goal to be completed. The change-goal-status operator is proposed when the robot is at the goal waypoint (the robot's location and orientation are within some thresholds of the goal state). The application of the change-goal-status operator changes a goal's status from active to complete. When the status of a turn goal is changed to complete, the most recent Hough information is removed and the compass direction of the robot is updated to reflect the completed turn. The change-plan-status is proposed after the last goal in a plan has been completed and the topological map has been updated (if necessary); its application changes the status of a navigation plan from active to complete and sends a command to the output-link to reset the occupancy grid.

The calculate-goal-state operator uses available knowledge about the environment, including the width and orientation of the previous hall, the distance between the robot and the wall in front of it (if a wall is present), and the robot state, to calculate the desired waypoint state of an active goal. The desired waypoint state is defined by an x-y location and an orientation. The reasoning the agent performs to calculate these waypoints could be easier if additional information about the intersection width and depth could be measured with additional sensors. When possible, the orientation of a goal waypoint is determined using the orientation of the current hall. This helps reduce problems with setting an undesirable goal orientation if the robot is not well aligned with the hall. For situations where there may not be an up to date hall orientation available, the desired amount to turn is added to the robot's current orientation to set the orientation of the desired waypoint.

The operator for calculating desired waypoint locations for go-straight goals has many sources of information to take into consideration. When the agent can take advantage of knowing there is a wall in front of the robot (e.g. the first goal in a turn right plan in a 'T' intersection), it uses information from the front sonar distance to set the location of the waypoint in front of the robot. However, when there is not a wall in front of the robot (e.g. the first and third goals of a turn

right plan in a ‘Tl’ intersection), the agent uses a percentage of the hall width to set the location of the waypoint in front of the robot. The agent can also consider if a navigation plan has just been interrupted, as there are situations where the robot may already have made progress towards the middle of the intersection that should be considered when setting the goal waypoint.

Going to a goal waypoint: When the Soar agent has an active goal with a goal waypoint and the agent is not at the desired location and orientation of that waypoint, it uses the `goToWaypoint` abstract operator to move to the waypoint. When the robot is at the desired waypoint (as defined by thresholds for the distance to and orientation of the waypoint), the goal is considered to have been achieved. Since the robot is able to turn in place, the agent first moves the robot to the waypoint, where it can turn to match the orientation if necessary. There are also operators in the `goToWaypoint` subgoal that can change the status of goals and plans if the agent encounters unexpected obstacles while trying to reach a goal waypoint.

Moving to the waypoint location: The move-to-waypoint operator is proposed if the robot is not at the location of the waypoint. In the move-to-waypoint subgoal, productions are used to calculate the distance to the desired waypoint, the orientation to the waypoint location, and if the robot is aligned with, facing to the right of, or facing to the left of this desired orientation.

There are three operators (`go-straight`, `turn-to-waypointLoc`, and `wait`) used to place motor commands on the agent’s output link. If the robot is aligned with the waypoint and there is not an obstacle in front of the robot, a `go-straight` operator is proposed and motor commands are placed on the agent’s output-link when this operator is applied. If the robot is aligned with the waypoint and there is an obstacle in front of the robot, the agent proposes a `wait` operator to allow the obstacle to be detected in the occupancy grid and the intersection detection algorithm. The `turn-to-waypointLoc` operator, which is proposed if the robot is not aligned with the waypoint it is attempting to reach, attempts to turn the robot until it is aligned with the waypoint.

There are two sets of conditions in which the `noteIntersection` operator is pro-

posed in the move-to-waypoint subgoal. Both cases include the usual conditions for noting an intersection: the robot has moved far enough away from the last location the occupancy grid was initialized or reset, the potential walls identified by the Hough transform are aligned with the robot, and the intersection type with the highest confidence value has been identified as a good detection. The last condition for the first case is that the agent is executing a go-straight goal in a turning plan before the turn goal in the plan has been completed (e.g. the first goal of a turn right plan in a ‘R’ intersection). The last condition for the second case is that the agent is executing a go straight plan (so there is only one goal, which is going straight) and the robot has traveled less than half the distance to the current goal waypoint.

In addition to the `noteIntersection` operator, the `removeIntersection`, `updateHoughInfo`, `updateMapIntersection`, and `updateMapObject` operators are used in the `goToWaypoint` subgoal. There are also operators that can interrupt the current plan if the senses an unexpected obstacle at a point in the plan where the agent can no longer note intersections. These operators prevent the robot from colliding with obstacles and the topological map from being updated with incorrect intersections.

Turning at a waypoint: The turn-to-waypoint operator is proposed if the robot is at the location of the waypoint, but is not aligned with the waypoint’s specified orientation. There are three operators that can be used to send motor commands to the Java program: the turn-left operator is proposed if the robot is facing to the right of the waypoint, the turn-right operator is proposed if the robot is facing to the left of the waypoint, and the wait operator is proposed if there is an obstacle in the direction that the robot has to turn to reach the waypoint. The `noteIntersection` operator is proposed for same two sets of conditions for which it can be proposed in the move-to-waypoint subgoal.

Stopping a plan for a side obstacle: While the agent is attempting to complete the third goal of turning plan (that is, trying to move straight to get out of intersection and into the hall leaving that intersection), if the robot is not aligned with the new hall (usually due to errors in odometry that accumulated

while the robot turned to reach the second goal's waypoint), the robot could move too close to a wall on one of its sides while it is trying to reach the goal waypoint. The agent can use its three most recent side sonar measurements to recognize this situation. When this happens, the agent assumes that the robot is in the new hall, but not well aligned with the orientation of the new hall and proposes the `stopPlanSideObstacle` operator. This operator changes the status of the third goal in the navigation plan from active to complete, allowing the agent to use the `navigate` operator to move the robot to the middle of hall.

Confirming an intersection: As previously described, the purpose of the first goal of a plan whose action is to turn left or right in an intersection is to move the robot to the center of the intersection and to confirm that the intersection detection that caused the navigation plan to be generated was correct. If the robot has reached the waypoint location of the first goal in a turn left or turn right plan and the agent can no longer detect the intersection that caused the plan to be generated, the `confirm-intersection` operator is proposed. The `confirm-intersection` operator uses a `wait` operator to keep the robot stationary to allow the agent some additional opportunities to detect the triggering intersection before assuming the triggering intersection was incorrectly detected and interrupting the plan. Alternatively, if the agent is able to detect the intersection type that caused the current plan to be generated, the conditions that led to the `confirm-intersection` proposal will no longer be true and the execution of the navigation plan can continue.

Updating the topological map: When the last goal of a navigation plan (that has been generated as the result of an intersection detection) is complete, the Soar agent adds information about that intersection to the topological map. Operators are used to:

- identify each node in the topological map with a unique number
- record the type of intersection that will be detected when entering node from each possible direction
- identify which intersections that have been recorded by the `record-intersection` operator correspond to each node

- record information about the connections between the current node and other nodes in the topological map, including:
 - the node numbers of and direction to any connected nodes or important places (like the start of the test)
 - the distance between connected nodes (a distance that is updated every time the connection is traveled by the robot).

6.3.5 Return home

Once all objects of interest have been detected, the Soar agent uses the `returnHome` operator and information from the topological map to move the robot back to the location at which the test was started. The abstract operators used in the `returnHome` subgoal are shown in Figure 6.7.

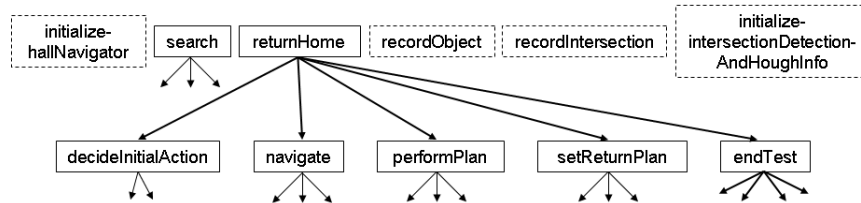


Figure 6.7: The abstract operators for the `returnHome` operator.

6.3.5.1 Decide initial return action

The first problem considered in the `returnHome` abstract operator is to decide how to get back to a previously visited topological map node to take advantage of the information about the environment stored in the topological map. If the agent is performing a navigation plan, it will attempt to complete this plan. For a turn around plan in a dead end, completing this plan should result in the robot heading towards a previously visited intersection. Completing navigation plans in other intersection types will permit additional information to be added to the topological map.

If the agent is not currently performing a plan, it uses information from the topological map to determine if the robot is moving towards a previously visited

area. If the robot is going to a previously visited area, the agent makes the decision to continue moving in its current direction using the navigate operator. If the robot is not going to previously visited area, the agent must turn the robot around to get the robot to a previously visited node. Since the agent will reset the occupancy grid after completing a turn plan, before starting to turn around the agent needs to make sure the robot is far enough away from the last intersection so that the robot will travel a large enough distance after turning around to generate an occupancy grid that will be accurate enough to reliably detect walls and intersections by the time the robot re-enters the most recently visited intersection. If the robot is far enough away from the last intersection, the agent makes the decision to turn around and sets a turn around plan. If the robot is not far enough away from the last intersection for the agent to turn around and reliably detect the intersection after turning around, the agent delays making the decision and begins to monitor the distance the robot is away from the last intersection while using the navigate operator. Once the robot is far enough away from the last intersection, the agent uses the `makeInitialDecision` operator to make the decision to turn around and generates a plan to turn around.

6.3.5.2 The navigate and perform plan operators

The navigate operator for the `returnHome` operator is the same as the navigate operator for the search operator. There are only a few small differences between the `performPlan` operator for the `returnHome` subgoal and the `performPlan` operator for the search subgoal. These differences are in operators used to update the topological map. In the `returnHome` subgoal, there is an additional operator that can be used to record a connection to the location where the robot turned around to return home if this initial decision was made somewhere besides an intersection. There is also an operator that recognizes when the robot has returned to the first node the agent encountered after beginning the test. The application of this operator prompts the agent to monitor the distance it has traveled from this node to the start location it is trying to reach.

6.3.5.3 Setting plans in intersections

While there are some similarities between the operators used to set plans in the `returnHome` and search abstract operators, operators used to set plans in the `returnHome` operator have the benefit of having a specified location to which the agent is trying to move (the location where the test began). The same operators described in Section 6.3.4.2 are used to determine if a plan should be interrupted or continued when a new intersection type is recorded while the agent is performing a plan generated as a result of a previously recorded intersection. When considering if the robot has been to its current intersection before, the agent considers the situations described in Section 6.3.4.2 as well as if the agent has completed a plan in which the robot turned around to begin returning to the start location. The operators used to set a plan to turn around in a dead end are also similar to those described in Section 6.3.4.2.

However, the operators used to set plans in intersections other than dead ends are different for the `returnHome` abstract operator than the operators used in the search abstract operator. Although the agent attempts to travel back to the start location using nodes and connections in the topological map that it has already visited, there is still the possibility that the agent detects an intersection that has not been recorded in the topological map. If an unknown intersection is detected, it is likely that the agent has not correctly detected the intersection. For example, if an unknown ‘Tl’ intersection is detected, it may be the intersection is really an ‘R’ that has not been detected because the front wall is still too far away to be detected. Although it may be worthwhile to change how the agent acts when unexpected intersections are detected, the current agent sets a plan and uses the mechanisms described for the search operator to later interrupt the plan if the plan is not able to be completed. As in the search operator, when setting a plan at a previously unvisited intersection in the `returnHome` operator, all relevant plan actions for the current intersection type are proposed with indifferent preferences (except for a plan to turn around, which is proposed with a worst preference) and the Soar decision procedure is used to select one plan to generate.

When the agent detects that the robot is currently in a node on the topological map, the agent takes advantage of Soar’s planning mechanism to select a plan that will lead to the start location. The agent proposes all plans that permit the agent

to follow previously recorded connections to adjacent nodes on the topological map (i.e., if there is not any information about a connection to the right of the current node in the topological map, the turn right plan will not be proposed even if the type of intersection detected means that a turn right plan is possible). These plans are proposed with acceptable, but not indifferent preferences (the turn around plan is still proposed with a worst preference). When there is more than one operator for creating a plan proposed with only an acceptable preference, the Soar agent does not know which operator to select and a operator tie impasse is generated.

In the subgoal to resolve the operator tie impasse, the Soar agent uses a set of rules (distributed with the Soar software) that is capable of performing planning to attempt to evaluate the result of selecting a tied operator. To use this set of rules, the Soar agent must have rules that define an abstraction of the agent's state that includes the information needed for planning, the state the robot is trying to reach, and any states that represent failed plans. For planning to return to the start location, the agent state is described by the number of the node the robot is currently in and the compass direction the robot is facing. The goal state is defined to be when the robot is at the start location (denoted by node zero) and failure states are defined to be states where the current node is a dead end or states that have been encountered previously during planning (indicating that the agent has encountered a loop in its planning).

For each of the tied operators, an evaluate-operator is proposed that attempts to determine if the operator will be useful in accomplishing the goal of returning to the test's starting location. Currently, the Soar agent selects the evaluate-operator to evaluate one of the tied operators at random, but it is possible to reason about which evaluate-operator should be selected (using operator preferences), allowing a more sophisticated search (such as A^*) to be performed. When an evaluate-operator is selected, the Soar agent uses an operator that internally simulates the action of the plan that is being evaluated on a copy of the abstraction of the agent's state.

For example, when the evaluate-operator corresponding to a turnLeft operator is selected, the agent simulates the results of a turn left plan by changing the copy of the agent's state (its compass direction and node number) to be that of the connected node that is to the left of the current node. When the application of

a simulated operator leads to a success or failure state, the symbolic evaluation of the operator that led to the state is generated as success or failure. When an evaluation (or evaluations) has generated enough information to select an operator from the set of tied operators, the operator tie impasse is resolved and the agent selects one of the tied operators.

If the application of a simulated operator does not lead to a success or failure state, operators are proposed corresponding to the possible plans at the agent's current simulated state. Proposal of these operators could lead to another operator tie impasse, resulting in the use of additional simulated operators to perform an internal search. This means if there is a series of impasses, there can be many copies of the robot state that can be used to internally simulate operators. This process of generating impasses to evaluate possible operators continues until the agent has generated enough information to select an operator from the set of tied operators. This planning is only used to select the plan for the current intersection, not for generating a plan the agent will blindly execute for the rest of the test.

When the Soar agent is evaluating operators during planning, it turns on Soar's chunking mechanism. This learning mechanism stores the results of planning as a Soar rule called a chunk. If the agent encounters the situation that has previously resulted in an operator tie impasse and planning, the agent may be able to use a chunk it previously learned to evaluate an operator instead of using an internal simulation.

6.3.5.4 Ending the test

Upon returning to the node closest to the location where the test started, the agent sets a navigation plan to move the robot through this intersection that will result in the robot entering the hall where the test was started. Once this navigation plan has been completed, the agent begins to monitor the distance the robot has traveled from this node to the start location (as described in Section 6.3.5.2). When this distance is greater than the length of the connection between the start location and the first node of the topological map, the robot should be near the location where the test was started and the endTest operator is proposed.

In the endTest operator, the agent generates instructions to the objects of interest found during the test using the nodes of the topological map as landmarks.

First the agent uses information about the location and time at which objects of interest were found to place the objects relative to topological map nodes, resulting in each object of interest being marked as being in or near a node of the topological map. Once each object has been placed relative to a node, operators use the same planning and internal simulation methods described in Section 6.3.5.3 to generate a path from the start of the test to the object of interest using the connections on the topological map (currently there are only Soar rules to do this for one object of interest). After a path has been generated, the path is broken up into directions that describe what action should be taken at each intersection to get from the start to the object of interest. Each step of these directions is placed on the agent's output-link to be retrieved by a Java program.

6.4 Results

Tests of the CRS developed for indoor search were performed in two environments and results obtained during these tests will be described here. Section 6.4.1 describes how the CRS uses the Soar agent to generate robot behavior. Sections 6.4.2 and 6.4.3 describes the results of tests performed using the SuperDroid. Section 6.4.4 describes a simulated test that was performed to show the usefulness of Soar's planning and chunking mechanisms in a topologically interesting environment.

6.4.1 Integration of Soar with the CRS

This section describes how Java is used in the CRS to communicate with and control the execution of a Soar agent (using the Soar Markup Language (SML)). Figure 6.8 shows the procedure a Java program uses to:

- obtain information about its environment,
- share that information with the Soar agent,
- generate a decision by the Soar agent,
- retrieve commands from the Soar agent,
- and process and carry out those commands.

The box in the lower left of the figure represents information about the environment obtained from sensors and as a result of sensor processing algorithms. The information generated in this box (by processes on the BrainStem controllers and by Java objects) is placed on the Soar agent's input link using the SML. The five phases of the Soar decision cycle discussed in Chapter 2 are represented by the input box (input phase), the box with the dashed border (the proposal, decision, and application phases), and the output box (output phase). A small part of the operator hierarchy generated in the Soar agent by productions is shown in the dashed box. While the Soar architecture uses the separate proposal, decision, and application phases to fire productions, the CRS runs Soar for one decision cycle at a time so there is not any interaction between the Soar agent and the Java program in between any of the five cycles. In other words, once information is placed on the Soar agent's input link and Java tells the Soar agent to run for a decision cycle, it does not communicate with the Soar agent until the agent has executed its five phases, when it checks to see if the Soar has placed any new information on its output link.

Figure 6.8 also shows how the presence and processing of commands affects the execution of the Java program in the CRS. If there is not a new command on the Soar agent's output link, the Java program tells the Soar agent to execute another decision cycle. If the Soar agent has generated a command other than a motor command, the Java program will process and execute this command before executing another Soar decision cycle. If a motor command has been sent to Java, that command will be processed and performed (using the motor control program on the BrainStem microcontroller). Additionally, new sensor measurements will be taken, the processing represented in the box in the lower left corner of the figure will be performed, and the appropriate information will be placed on Soar's input link before another Soar decision cycle is executed.

Similar to Figure 6.8, Figure 6.9 shows the interaction between Java and Soar in the CRS, but it includes more of the operator hierarchy in the dashed box representing the proposal, decision, and application phases of Soar's decision cycle. Figures 6.10–6.12 show the order of steps taken by the CRS as an indoor search test begins. The current location of the processing in the CRS in each figure is indicated by a red oval. First, sensor measurements are obtained and sensor processing

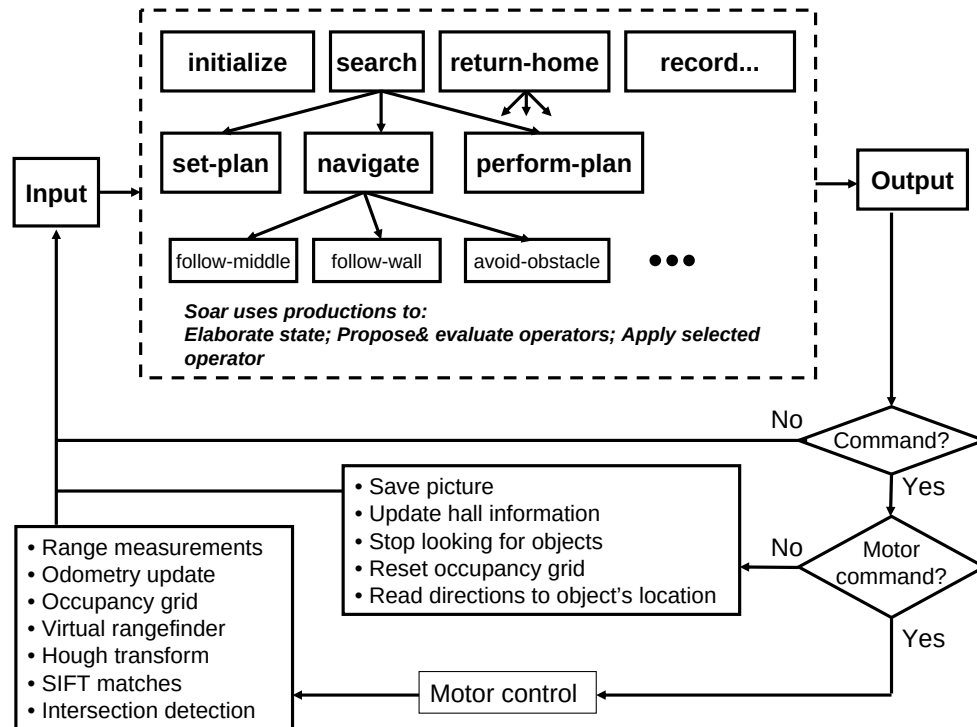


Figure 6.8: CRS and use of the Soar decision cycle

algorithms are used to generate information about the agent's environment (Figure 6.10a) and the information listed in Section 6.3.2 is placed on the Soar agent's input-link and the Java program tells the Soar agent to execute for one decision cycle (Figure 6.10b). Information placed on the input-link is retrieved by the Soar agent during its input phase. The Soar agent selects and applies the initialize-hallNavigator operator, but does not place any information on its output-link.

Since there were no commands on the agent's output-link for the Java program to process (Figure 6.10c), the Soar agent is executed for another decision cycle (Figure 6.10d). There is no new information for the Soar agent to get from its input-link. The agent selects the search operator, but again does not place any commands on its output-link. Again, there are no commands on Soar's output-link (Figure 6.10e). The Soar agent is executed for another decision cycle (Figure 6.10f). There is no new information on Soar's input-link. The agent selects the navigate

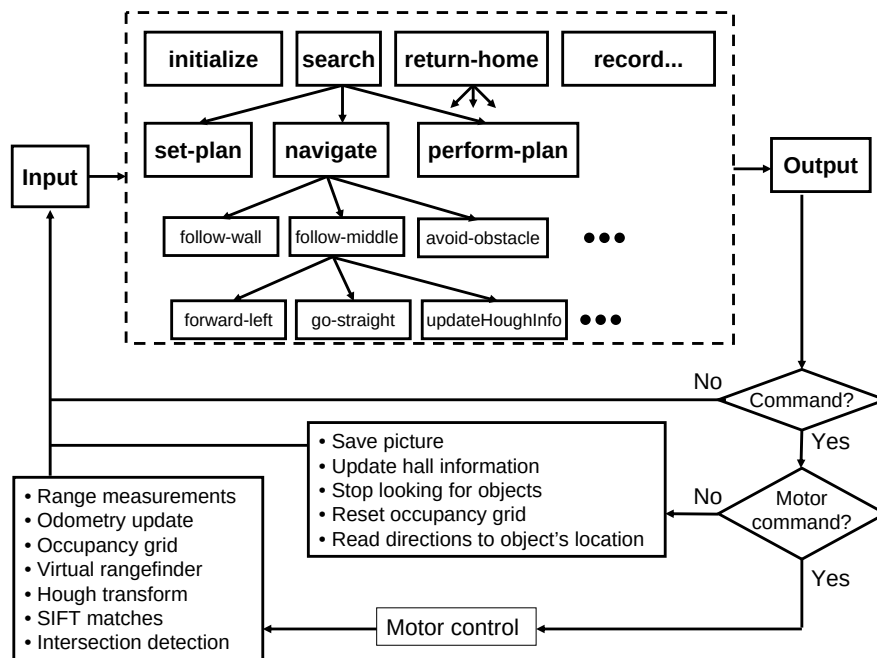
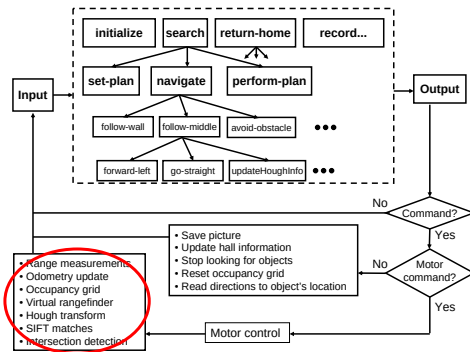


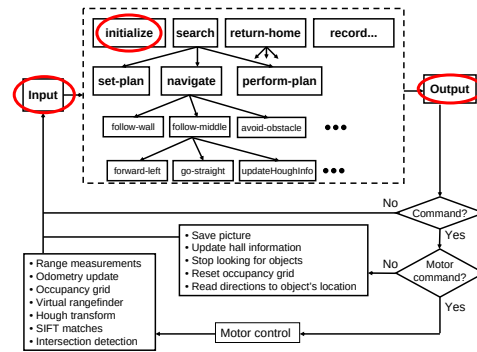
Figure 6.9: CRS and use of the Soar decision cycle

operator, moving down the operator hierarchy. The Soar agent can not apply the navigate operator directly, so no commands are placed on the agent's output-link. With no commands to execute (Figure 6.11a), the Java program executes another decision cycle in Soar (Figure 6.11b) in which there is no new information on the input-link. The agent moves further down the operator hierarchy, selecting the followMiddle operator. The followMiddle operator is another abstract operator that can not immediately generate commands, so again no command can be placed on the Soar agent's output-link and there is not a command for the Java program to process (Figure 6.11c).

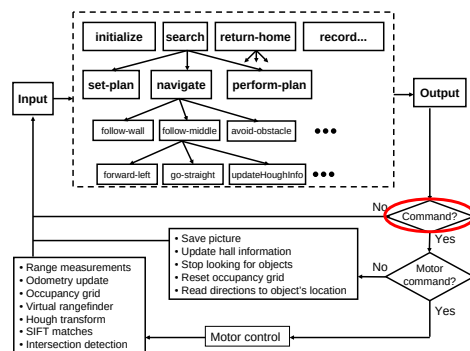
The Soar agent is executed for another decision cycle (Figure 6.11d). There is still not any new information on the agent's input-link. The Soar agent has moved far enough down its operator hierarchy to select the go-straight operator, a primitive operator that is capable of generating a motor command, which is placed on the output-link. The Java program retrieves the command from the output-link (Figure 6.11e), recognizes it as a motor command (Figure 6.11f), and sends the



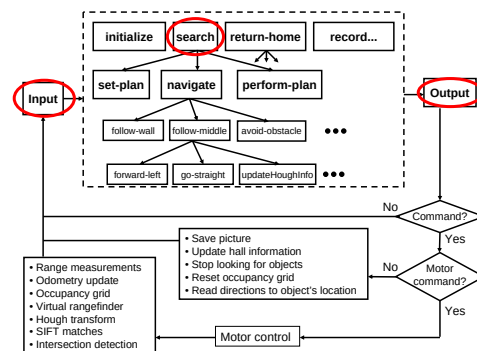
(a)



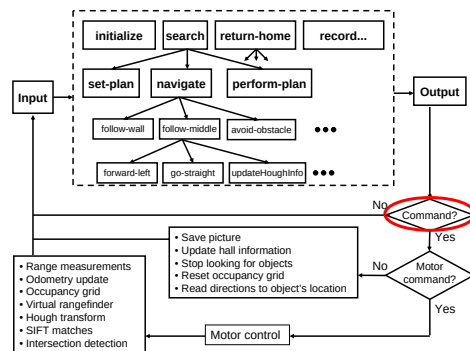
(b)



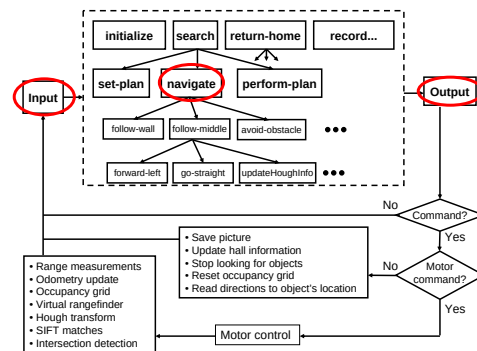
(c)



(d)



(e)



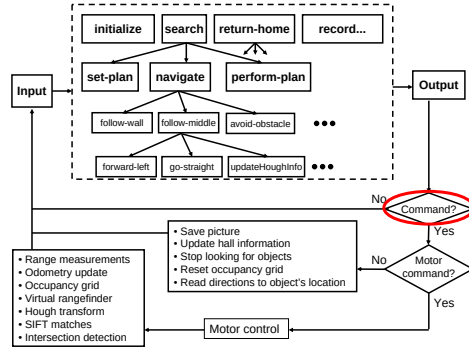
(f)

appropriate information to the CRS's motor control program on the BrainStem microcontroller (Figure 6.12a).

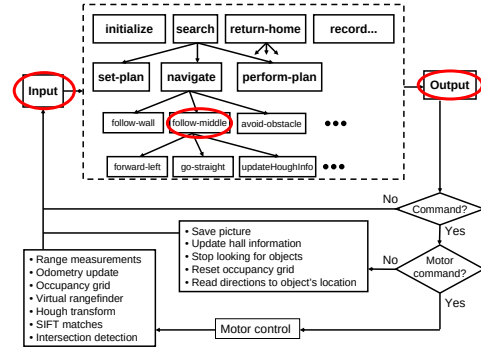
The CRS then obtains new information from its sensors, generates information from its sensor processing algorithms, and places the relevant new information on the Soar agent's input-link (Figure 6.12b). When the Soar agent is executed for another decision cycle (Figure 6.12c), it retrieves the new information from its input-link. Since the conditions that previously resulted in the Soar agent generating the operator hierarchy and selecting the followMiddle abstract operator are still true, the agent does not have to select operators to move down the hierarchy and can select the go-straight operator again. This operator places a motor command on the agent's output-link, which is retrieved (Figure 6.12d) and processed as a motor command (Figure 6.12e). The appropriate information is again sent to the motor control program on the BrainStem (Figure 6.12f).

The example described in Figures 6.10–6.12 focused on explaining the interaction between a Java program and Soar agent in the CRS. It also demonstrated that the problem solving used to generate an operator hierarchy can be saved from decision cycle to decision cycle, allowing the agent to remain in a lower level of the hierarchy if the conditions that resulted in the generation of the hierarchy is still true. However, this example did not give any details about how operators are proposed and applied within Soar's decision cycle. Figures 6.13–6.18 describe the details of what occurs in the individual phases of the Soar decision cycle at the beginning of an indoor search mission.

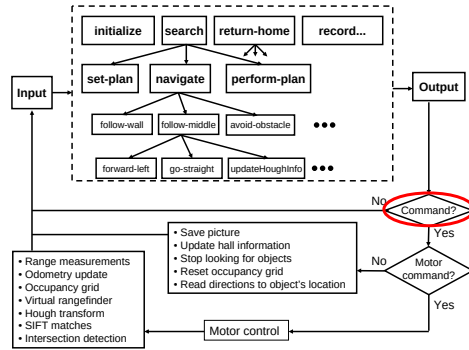
Each row in the tables shown in Figure 6.13–6.18 represents one decision cycle executed by the Soar agent. The green arrows in each of these figures represent the processing that occurs in Java, such as processing any commands placed by the Soar agent on its output-link, performing motor control, and generating new information about the environment to be placed on Soar's input-link for use in the next decision cycle. Phases without any text in them represent phases in which no Soar productions were fired or retracted (the propose and apply phases), no new information was retrieved from the input-link, or no new information was available on the output-link for the Java program to process. All of the processing performed by the Soar agent is not shown in these figures, but the changes in working memory and production rule firings and retractions that were most important for the



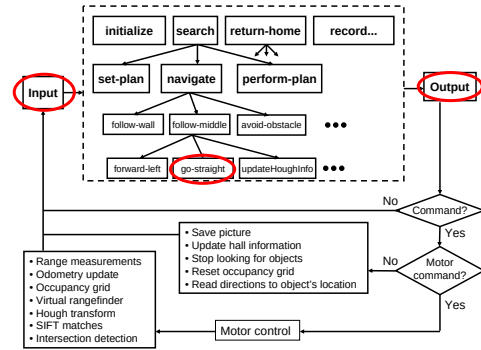
(a)



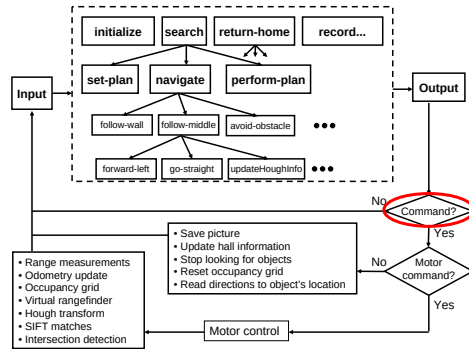
(b)



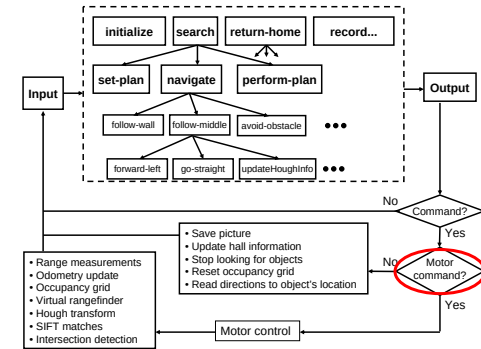
(c)



(d)



(e)



(f)

Figure 6.11: Figure 2 of 3 showing the interaction between the CRS and the Soar agent (continued in Figure 6.12).



Figure 6.12: Figure 3 of 3 showing the interaction between the CRS and the Soar agent.

decision making demonstrated in Figures 6.13–6.18 are shown.

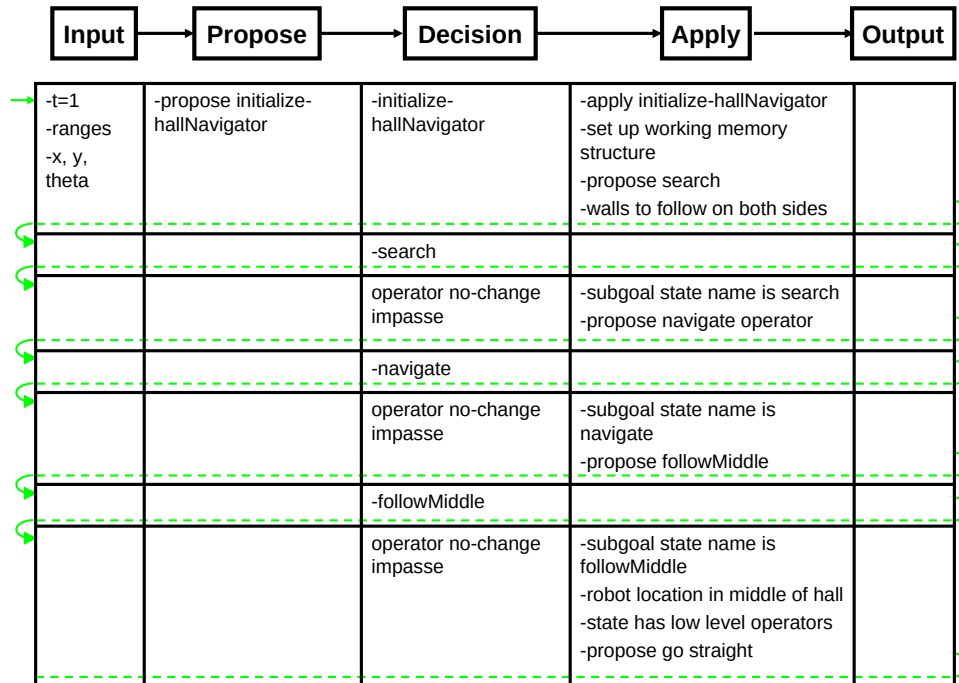


Figure 6.13: Figure 1 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.14).

The first row in Figure 6.13 represents the first decision cycle of the test. The sensor information for the first time step is retrieved from the input-link. The initialize-hallNavigator operator is propose, selected, and applied. Applying this operator sets up the structure of working memory. Since the agent has not detected the object of interest, the search operator is proposed. Using the range information on the input-link, the agent determines there are walls to follow on both sides of the robot. When all possible productions have fired and retracted and there are no further changes in working memory, the apply phase of the decision cycle ends. No information has been placed on the output-link, so, as shown by Figures 6.10–6.12, the Soar agent is executed for another decision cycle without any new information being placed on the input-link. Since there have been no changes in working memory since the end of the apply phase of the previous cycle, no productions are fired or retracted in the propose phase. The search operator proposed during the last decision cycle is selected, but since it is an abstract operator, the Soar agent

does not know how to apply it. An operator no-change impasse is generated in the decision phase of the next decision cycle (the third row of Figure 6.13). In the apply phase, a production fires to give the automatically generated subgoal the state name search. Once the state's name is search, the navigate operator is proposed. This navigate operator is selected in the decision phase of the next decision cycle. Since navigate is an abstract operator, another operator no-change impasse is generated in the next decision cycle. In the apply phase after this impasse occurs, the state name of the automatically generated subgoal is determined to be navigate. Then, since there are walls to follow on both sides of the robot (as elaborated in the first decision cycle), the followMiddle operator is proposed. The followMiddle is an abstract operator, so no commands are placed on the output-link and no new information is placed on the input-link by the Java program. Another operator no-change impasse occurs, generating a subgoal to implement the followMiddle operator. The name of the subgoal state is followMiddle. A production elaborates that the robot is in the middle of the hall. Another production notes that the followMiddle state has low level operators, which means that there are operators that place motor commands on the agent's output-link that can be applied in this state. Since, the robot's location is in the middle of the hall, the go-straight operator is proposed. The proposed go-straight operator is selected during the next decision cycle (the first row of Figure 6.14). The application of this operator (apply move) results in a motor command being placed on the output-link. An operator no-change impasse occurs because the application of the move operator does not result in the conditions of any additional productions being matched.

The Java program retrieves the motor command from the Soar agent's output-link, sends the command information to the CRS's motor control programs, sends updated information about the environment to the Soar agent, and adds a complete status attribute to the go straight motor command from the previous time step. In the third row of Figure 6.14, the agent uses the new range measurements from the sonar sensors to elaborate that the robot's location is in the middle of the hall. The go-straight operator is then proposed and selected. This operator is applied using the apply move operator and the motor command is placed on Soar's output-link. The processing shown for this decision cycle is repeated through the last row of Figure 6.14 (for the third, fourth, fifth, and sixth time steps). The conditions

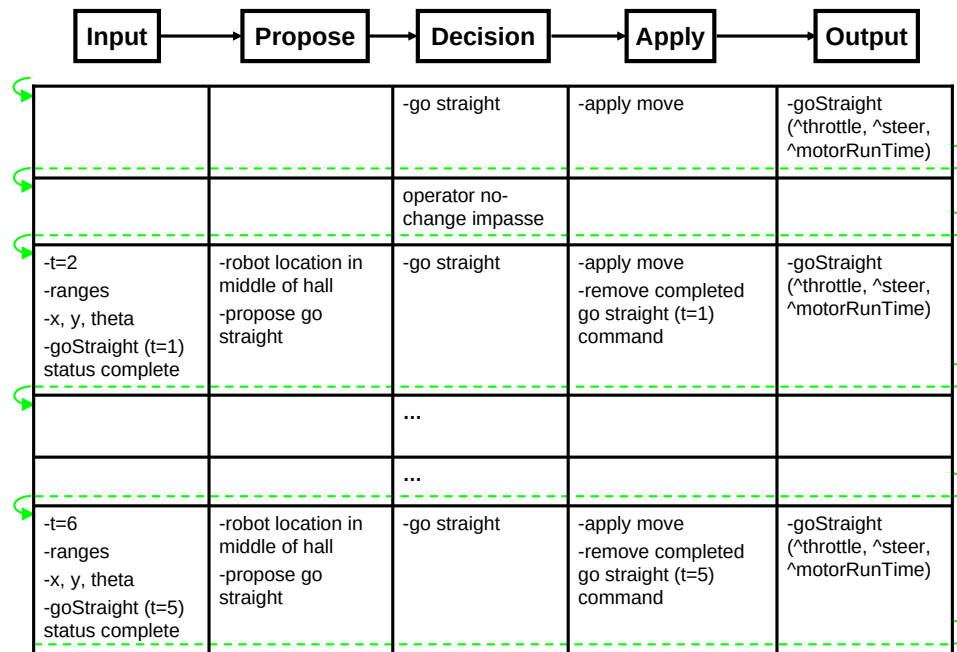


Figure 6.14: Figure 2 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.15).

that led to the proposal and selection of the search, navigate, and followMiddle operators during the first time step ($t = 1$) are still true, so the agent remains in the followMiddle subgoal.

After the input phase (shown in the first row of Figure 6.15) of the seventh time step ($t = 7$), the distance the robot has traveled from the initialization (or reset) of the occupancy grid is calculated to be large enough to propose initializing intersection detections and using the Hough information. (Note, this distance had been calculated during every time step since the beginning of the test; the distance had just not been large enough for the this operator to be proposed). This operator is proposed with a best preference. The robot is still elaborated to be in the middle of the hall, so the go-straight operator is proposed again. The operator to initialize the detection of intersections and use of information from the Hough transform is selected because of the preferences associated with its proposal compared to the preferences associated with the search operator proposed when $t = 1$. After this operator is applied, the search operator, which has remained proposed since $t = 1$

is selected. The rest of the decision cycles in Figure 6.15 show the regeneration of the operator hierarchy first established in the $t = 1$ timestep.

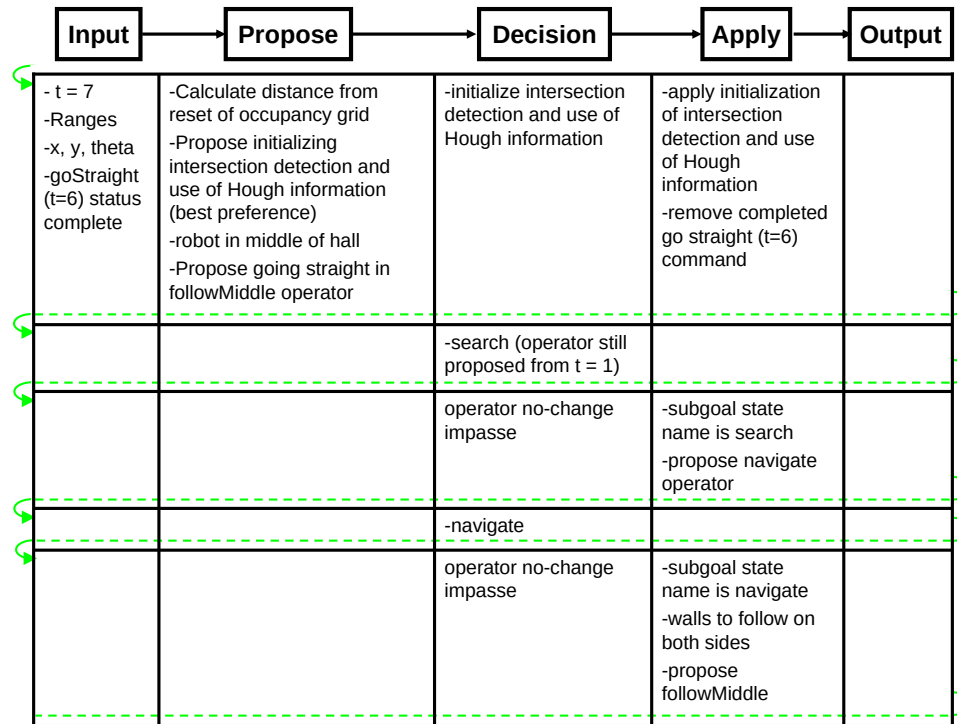


Figure 6.15: Figure 3 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.16).

Now, when the agent reaches the followMiddle subgoal (row two of Figure 6.16), the conditions of more productions can be matched since the use of intersection detection and Hough transform information has been initialized. Both the updateHoughInfo and go-straight operators are proposed and a production is fired to create a preference that the updateHoughInfo operator be selected before the go-straight operator. In the next decision cycle, the updateHoughInfo operator is selected and applied (including placing an updateHallWidth command on the agent's output-link). Applying this operator results in changes in working memory that cause the updateHoughInfo operator proposal to retract. Using information from the Hough transform, the agent is able to elaborate that the robot is aligned with its current hall and therefore propose a second go-straight operator with a production that tests that the robot is both aligned with and located in the mid-

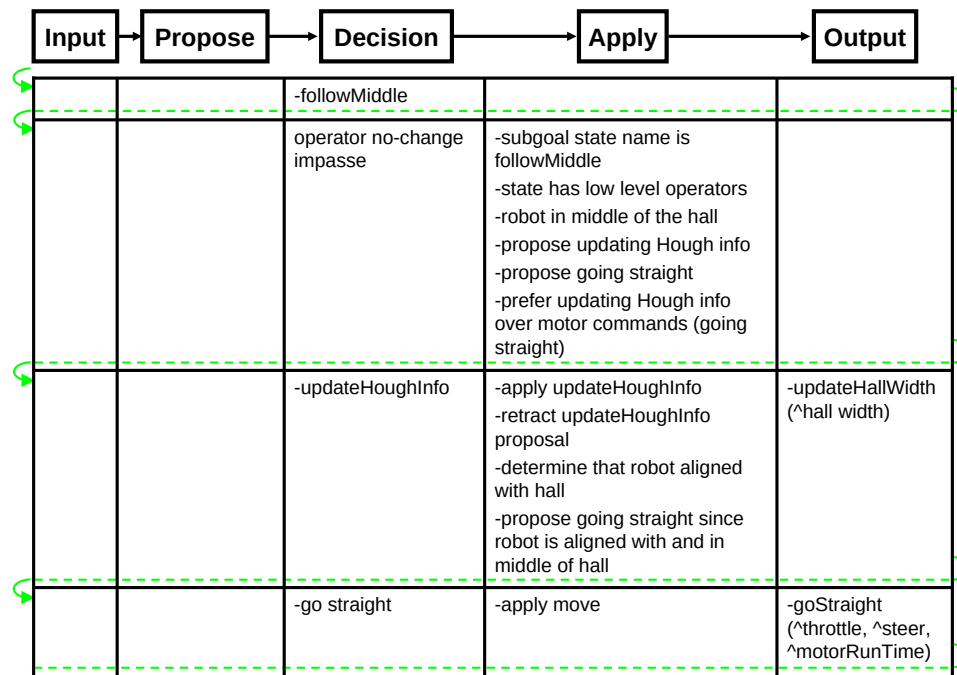


Figure 6.16: Figure 4 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.17).

dle of the hall. After the output phase of this decision cycle, the Java program processes the updateHallWidth command on the agent's output link, updating the width of the hall as a parameter for the fuzzy logic system used for identifying intersections. In the next decision cycle, the go-straight operator proposed in the last decision cycle is selected (over the go-straight operator proposed using only the fact that the robot location is in the middle of the hall) and applied.

Since the Soar agent has determined it can start using the confidence values from the intersection detection system, there are additional productions that the Soar agent can consider. At the beginning of the propose phase of the first decision cycle (bottom row of Figure 6.17) in the $t = 8$ time step, the agent's current state is the followMiddle subgoal. Productions fire that find the intersection type with the highest confidence value and determine that the robot is in the middle of and aligned with the hall. The production that proposed the go-straight operator in the previous time step is retracted and the updateHoughInfo operator is proposed. Two different go-straight operators are proposed using the knowledge that the

robot is in the middle of and aligned with the hall. Additionally, the intersection type with the highest confidence value is determined to be a good detection; the note intersection operator is then proposed. Productions then generate preferences that can be used to select an operator from the four proposed operators: the noteIntersection operator is preferred over the updateHoughInfo operator, which is preferred over the go-straight operators. Soar uses these preferences to select the noteIntersection operator. Once this operator has been applied, the conditions used to propose it are no longer all true and the production used to propose the operator retracts.

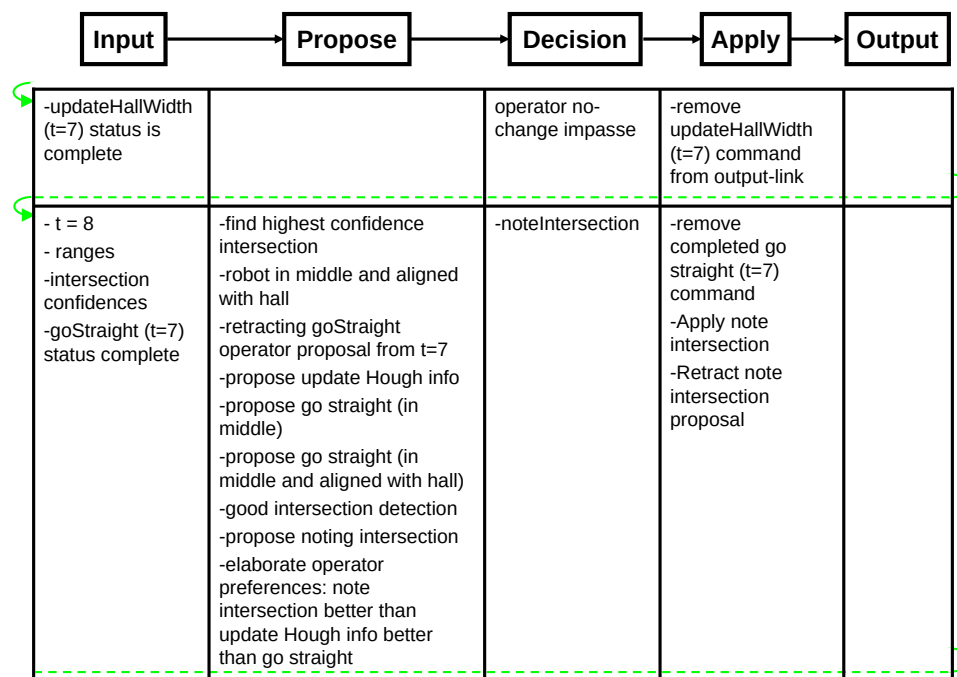


Figure 6.17: Figure 5 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission (continued in Figure 6.18).

In the next decision cycle (the top row of Figure 6.18), the updateHoughInfo operator is selected (it is still preferred over the go-straight operators because of the preferences generated in the last decision cycle). Applying this operator changes working memory such that the updateHoughInfo operator proposal conditions are no longer true, and the production that was used to propose this operator retracts. The updateHallWidth command is placed on the output-link for retrieval

by the Java program. In the next decision cycle, a go-straight operator can be selected (since the noteIntersection and updateHoughInfo operators are no longer candidates for selection). The go-straight operator is applied by the apply move production and a motor command is placed on the output-link.

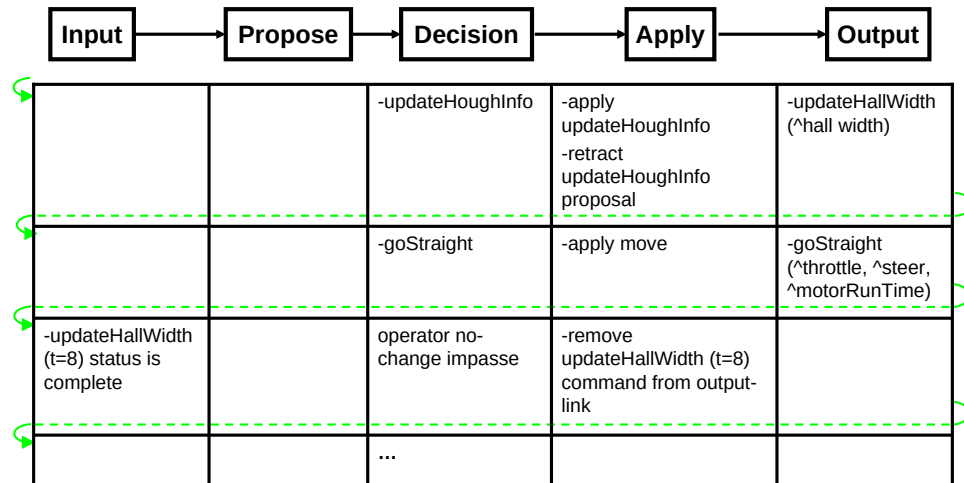


Figure 6.18: Figure 6 of 6 demonstrating the use of the Soar decision cycle at the beginning of an indoor search mission.

The example described in Figures 6.13–6.18 shows how operator no-change impasses and automatic subgoalting are used to hierarchically decompose the search operator within the framework of the Soar decision cycle. It also demonstrates how, within the same phase of a decision cycle, the changes in working memory that result from the firing of a production can lead to the conditions of an additional production matching and that production then firing. The propose phase in the decision cycle shown in the bottom row of Figure 6.17 reflects a phase in which this process repeats several times until quiescence is reached and no other productions can fire or retract. The agent was also able to maintain its operator hierarchy when possible by staying in the followMiddle subgoal from $t = 1$ through $t = 6$, select and apply a higher priority operator (initializing intersection detection and use of information from the Hough transform when necessary (at $t = 7$)), and then regenerate the operator hierarchy used to continue navigating down the middle of a hall.

Using the same format used in Figures 6.13–6.18 to show the phases of the Soar decision cycle and interaction between the Soar agent and the rest of the CRS, Figures 6.19–6.24 describe another example of the types of reasoning that can be performed using Soar. At the beginning of the example shown in Figures 6.19–6.24, the agent is using the search operator to look for the object of interest for this test (a doll) and is performing a plan to move through an intersection. Previous to the first time step represented in Figure 6.19, the doll has been detected and its location has been recorded.

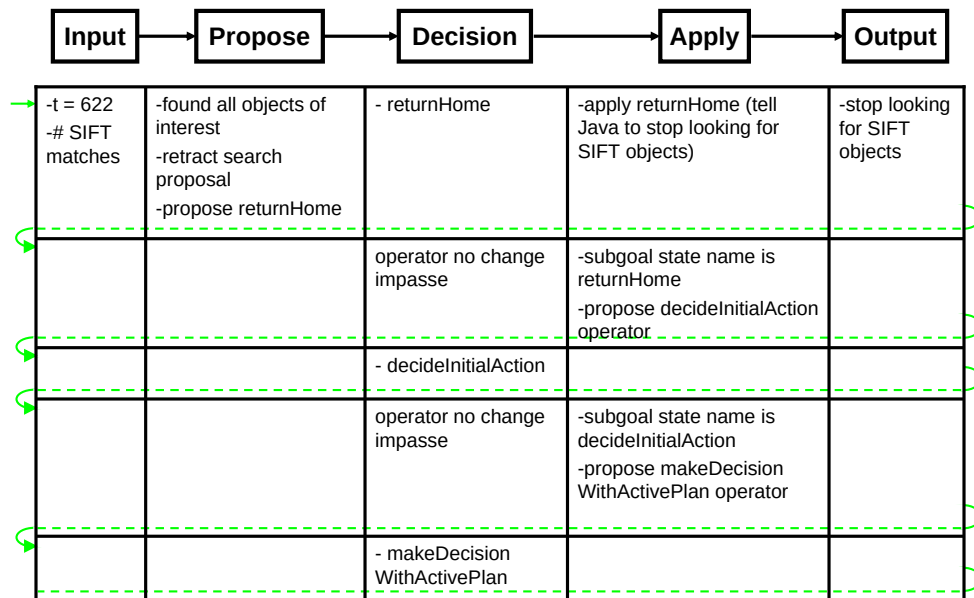


Figure 6.19: Figure 1 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.20).

At time $t = 622$, the number of SIFT features indicates that the agent can no longer sense the doll and the agent determines that all objects of interest (only the doll for this test) have been located. The creation of a working memory element signifying this information results in the retraction of the search proposal and the proposal of the returnHome operator. The returnHome operator is selected and applied, resulting in a command for the CRS to stop looking for SIFT objects being

placed on the output-link for retrieval by the Java program. Once the returnHome operator is selected, an operator no-change impasse occurs, resulting in a subgoal to determine the agent's initial action. The rest of Figure 6.19 and Figure 6.20 show the hierarchical decomposition of operators used to make this decision: as described in Section 6.3.5.1, since the agent is performing a plan to move through an intersection that is not a dead end, the agent decides to complete the execution of the plan. If the plan is able to be completed, a node representing the current intersection will be added to the topological map, which may be useful to more accurately describe the location of the object of interest.

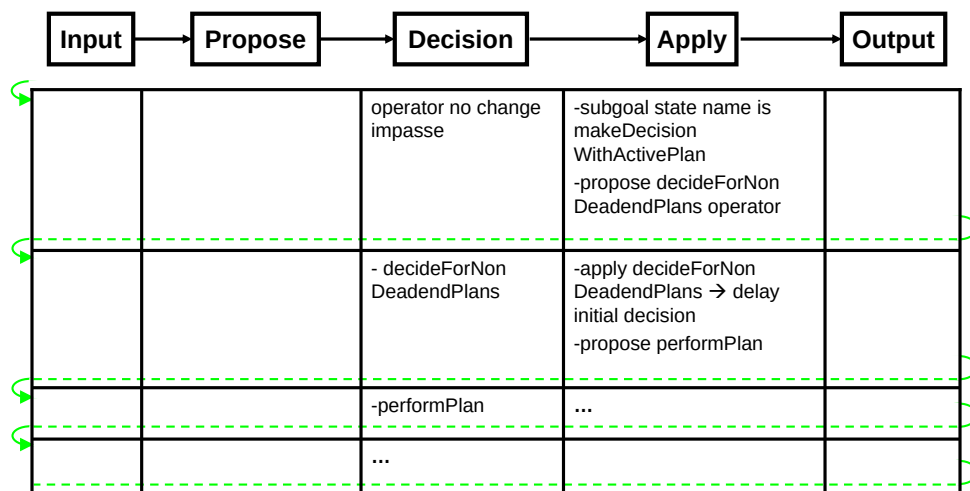


Figure 6.20: Figure 2 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.21).

As shown in Figure 6.21, a few time steps later ($t = 625$), the agent uses the robot's current state (x , y , and θ values) to elaborate that the robot is close enough to be considered to have reached the desired waypoint. Since the robot has reached the goal location, a production to change the status of the goal fires and the goToWaypoint operator proposal retracts. The change-goal-status operator is selected and applied, changing the status of the just completed goal from active to complete. Since the goal is no longer active, the i-supported information that

was generated because the goal was active (such as the distance from the goal) is removed from working memory when the productions that generated these elements are retracted. Since the just completed goal was the last goal of the current navigation plan, the updateMap operator is proposed.

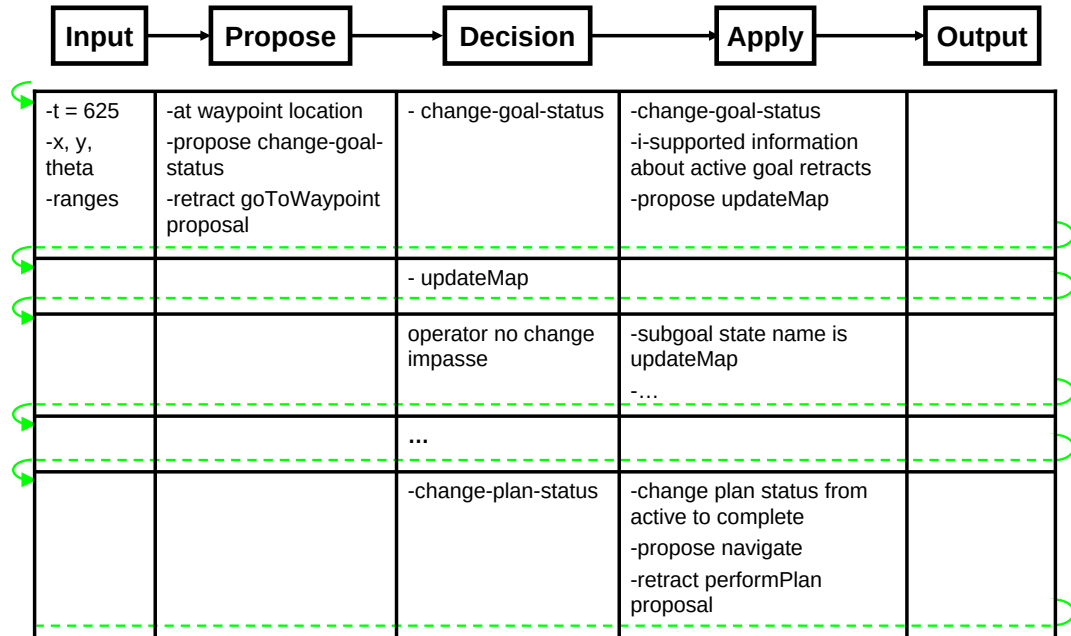


Figure 6.21: Figure 3 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.22).

The updateMap operator is an abstract operator, so an operator no-change impasse occurs, resulting in a subgoal that updates the topological map with information about the intersection the robot just moved through. The details of how the map is updated are not given here, but once the map has been updated, the change-plan-status is selected and the plan status is changed from active to complete in the apply phase of the decision cycle in the bottom row of Figure 6.21. Since there is not an active navigation plan, the navigate operator is proposed and the performPlan operator proposal that was selected to perform the just completed plan is retracted.

The navigate operator is selected and subgoals are generated to follow the wall located to the robot's right side (row two of Figure 6.22). During the next time step ($t = 626$), the agent is still using the followWall subgoal to navigate in its environment. The robot's current state is used to calculate the distance the robot has traveled since entering the navigate operator at $t = 625$. Productions are fired to elaborate that the robot is aligned with the hall, but too close to the wall it is attempting to follow. The updateHoughInfo and forward-left operators are proposed, with the updateHoughInfo operator being preferred over the forward-left operator. Additionally, a production fires to create a preference for selecting the navigate operator over the decideInitialAction (which is still proposed from $t = 622$ since the decision was delayed at that time) since the robot has not traveled far enough from where the navigate operator was selected in $t = 625$. The updateHoughInfo operator is selected and applied and then the forward-left operator is selected and applied in the next decision cycle.

After several time steps (at $t = 636$ in Figure 6.23), the agent again calculates the distance the robot has traveled since selecting the navigate operator at $t = 625$. The distance calculated is large enough so that if the robot turned around here, the CRS could generate an occupancy grid accurate enough to detect an intersection before the robot enters the intersection it left at $t = 625$. This new information causes a production to fire that creates a preference for selecting the decideInitialAction over the navigate operator. (The production used in previous time steps to prefer the navigate operator has retracted because it relied on an out-of-date value of the distance the robot had traveled since selecting the navigate operator in $t = 625$). Productions also fire to elaborate that the robot is aligned with, but too close to the wall it is trying to follow (the agent is now following the left wall). In the rest of Figure 6.23 and Figure 6.24, the agent decides to set a plan to turn around to go to an area it has previously visited and begins to execute this plan.

The example described using Figures 6.19–6.24 demonstrated the Soar agent reasoning about when to turn around after detecting the object of interest during an indoor search mission. During this example, two operators (the navigate and decideInitialAction operators) were proposed (and remain proposed over several time steps) and productions that compare these candidate operators were able to

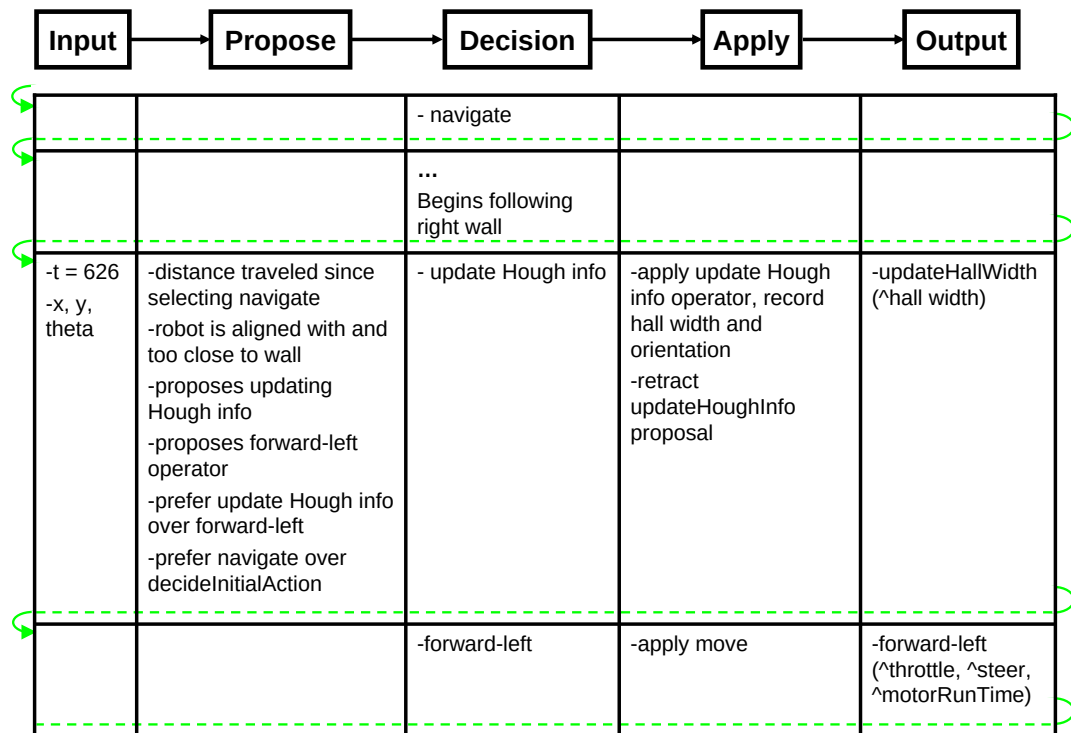


Figure 6.22: Figure 4 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.23).

change which operator was preferred at various times based on the specifics of the current situation. The limitation that the robot needs to travel forward before the occupancy grid becomes accurate enough to use is represented in a production that prefers that the robot moves away from an intersection (using the navigate operator) over deciding when to set a plan to turn around so the robot can begin to move towards the test's start location. Encoding this knowledge in a production is key to the successful behavior generated in this example and performing this type of reasoning in a reactive system would be very difficult.

6.4.2 Hallway tests

The tests discussed in this section were performed in a group of hallways on the first floor of the Hammond Building on the Pennsylvania State University campus.

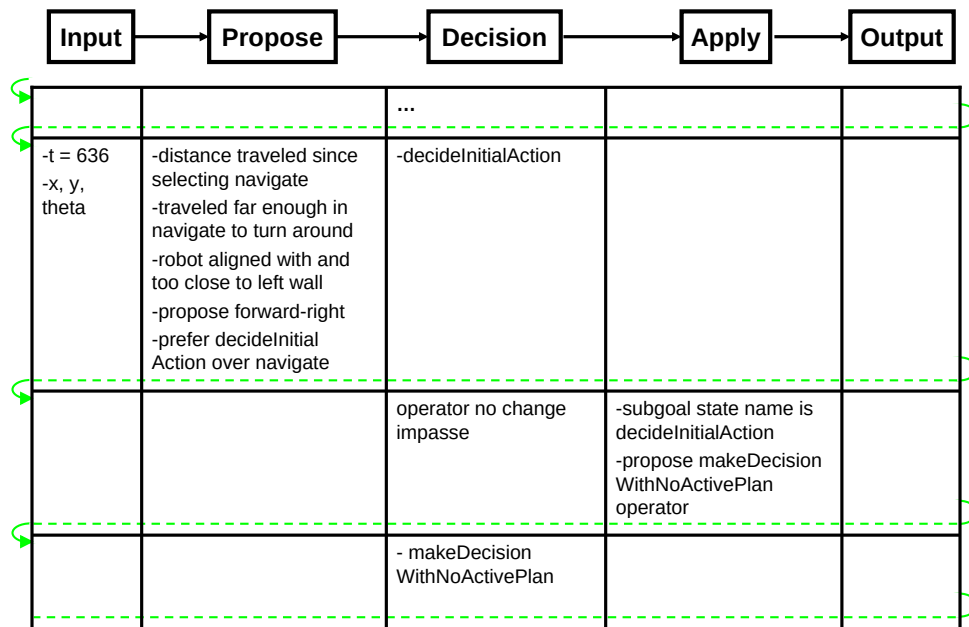


Figure 6.23: Figure 5 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator (continued in Figure 6.24).

The only difference in the systems used for these two tests was that the Soar agents had a slight difference in how a plan was selected when the robot entered an intersection that had not been previously visited. For the first test, the agent selected a plan at random from the proposed plans. For the second test, the agent preferred turn plans to go straight plans.

6.4.2.1 First test

Figure 6.25 shows the arrangement of the hallways in which these tests were conducted and the order in which the CRS searched the environment for the object of interest, a doll. In this figure, the numbered circles represent the locations where the Soar agent set plans that were able to be completed. The arrows represent movement by the robot between intersections; these movements are generally accomplished using the navigate operator, but, in a few cases, an incorrectly detected intersection results in some of the movement happening while the agent is trying

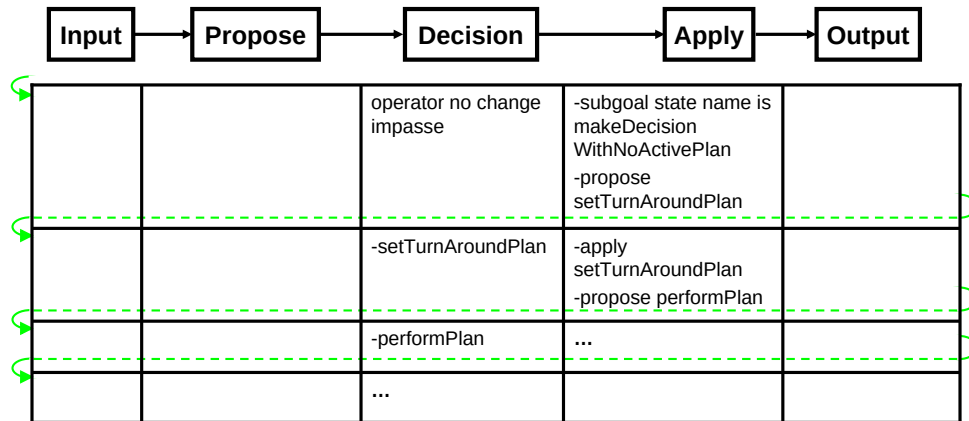


Figure 6.24: Figure 6 of 6 demonstrating the use of the Soar decision cycle when the agent decides its initial action in the returnHome operator.

to perform a navigation plan. A graphical representation of the topological map generated by the agent during this test is shown in Figure 6.26.

To start the test, the robot was placed facing north (towards the top of the figure) at the location represented by the green triangle and the doll was placed in the dead end of the side hallway near the bottom of the figure (represented by the black 'X'). The robot uses the navigate operator and detects a 'Tl' intersection at point 1 (node 1 in the topological map). Since the robot has not visited this intersection before, the agent selects the proposed go straight plan randomly over the proposed turn right plan. After completing this go straight plan, the agent again uses the navigate operator to move the robot to point 2 (node 2 in the topological map), where another 'Tl' intersection is detected. This time, the proposed turn right plan is randomly selected over the go straight plan. The robot performs the plan and enters the side hallway facing to the east. The robot navigates down the hall until a dead end is detected at point 3 (node 3), where the agent sets a plan to turn around.

When the robot reaches point 4, it is still too far away from the wall in front of it (the left most wall in the figure) for the intersection detection system to sense that wall. Therefore, the agent detects an 'X' intersection and proposes setting

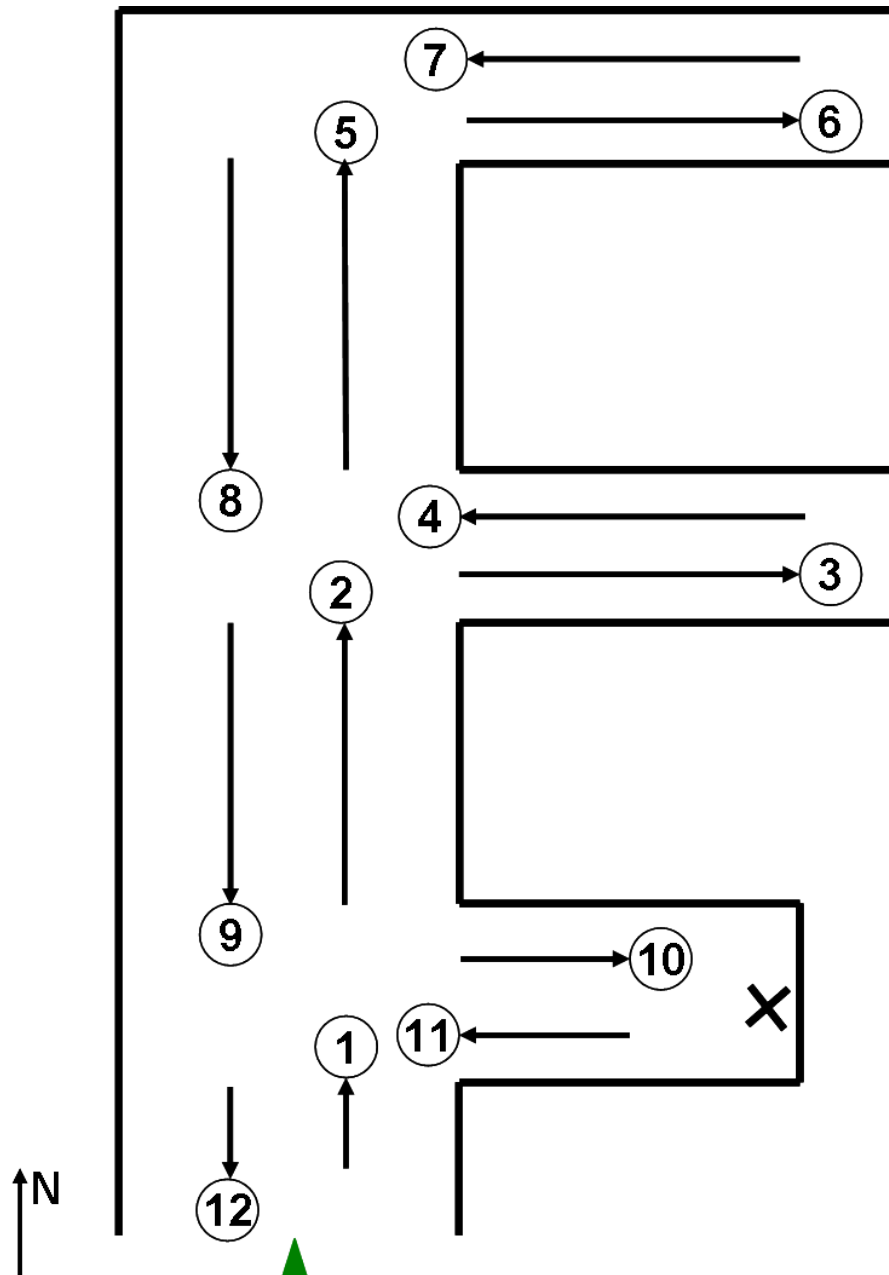


Figure 6.25: A map of the hallways used and arrows, labeled with numbers, indicating the order of movements the robot made through the environment during the test described in Section 6.4.2.1. The robot started at the green triangle and found the object of interest at the black 'X.'

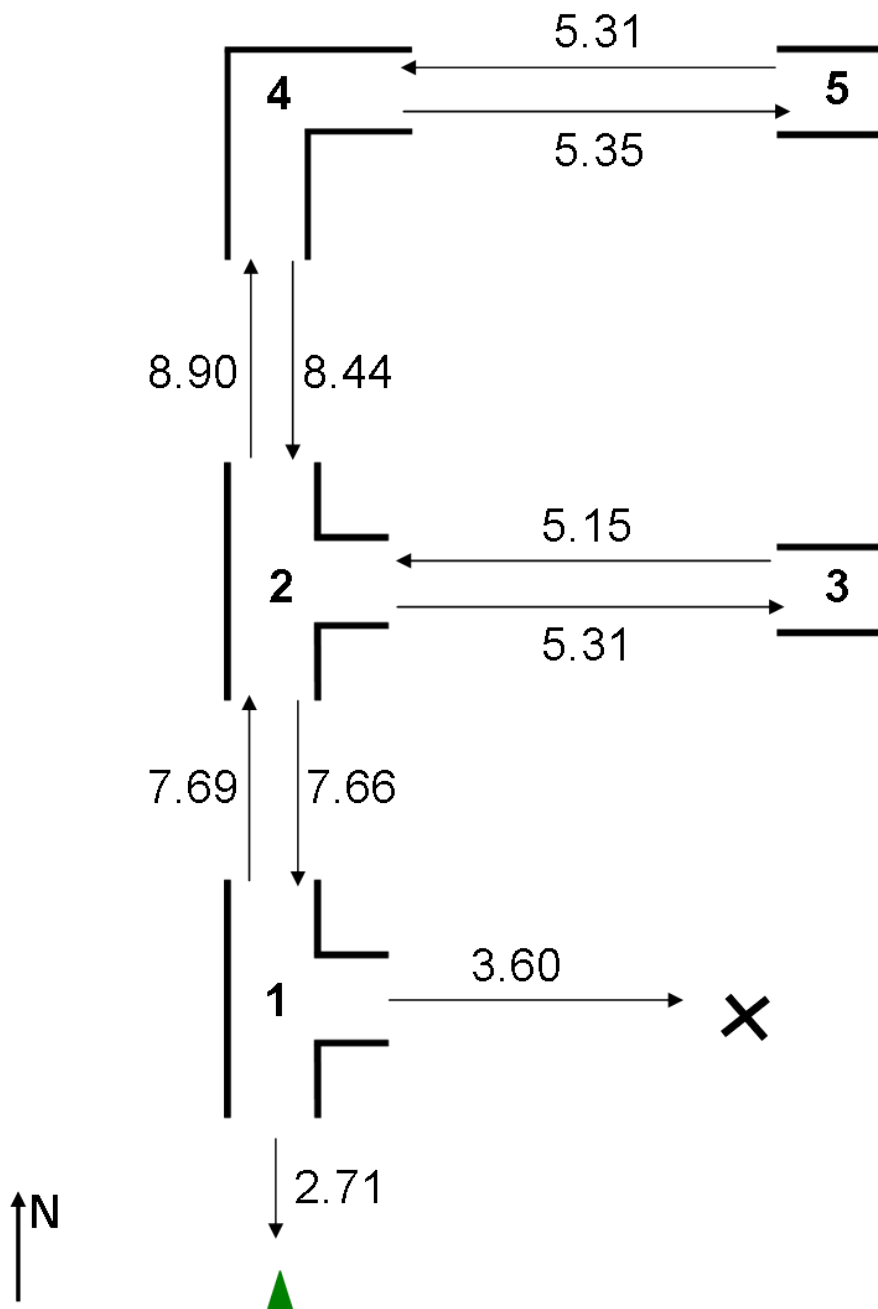


Figure 6.26: A depiction of the topological map generated by the agent during the test described in Section 6.4.2.1. The bold numbers 1–5 are the identification numbers given to each node in the map. A connection from one node to a second node is represented by an arrow pointing to the second node. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'

turn left, go straight, and turn right plans. The turn right plan is selected; while this plan is being performed, the robot moves close enough to the wall in front of it for the agent to detect a ‘T’ intersection. The turn right plan triggered by detection of the ‘X’ intersection is interrupted and operators to set turn left and turn right plans are proposed. Since the agent realizes that it has already visited this intersection before, a production fires that prefers the operator that will set a plan leading in a direction that the current node (node 2) of the topological map does not have a connection (the turn right plan) over the operator that will set a plan leading in a direction that the current node of the topological map does have a connection (the turn left plan).

After completing the turn right plan generated at point 4, the robot navigates to point 5 (node 4) where the agent detects a ‘R’ intersection and sets a turn right plan. The robot navigates down the side hallway until a dead end is detected at point 6 (node 5) and then turns around. At point 7, the agent detects a ‘Tr’ intersection (the wall in front of the robot is too far away) and sets a plan to turn to the left. During the execution of this plan, the front wall is sensed and the agent detects the ‘L’ intersection (node 4). The turn left plan triggered by the ‘Tr’ intersection is interrupted and a new plan to turn left to move through the ‘L’ intersection is set.

At point 8, the agent detects a ‘Tr’ intersection and recognizes that the robot has visited this intersection before and that this intersection corresponds to node 2 of the topological map. Operators to set turn left and go straight plans are proposed. The agent uses its productions that prefers a plan that will lead to a previously visited intersection that is not a dead end over a plan that will lead to a previously visited intersection that is a dead end to select the operator to set the go straight plan (over the operator to set a turn left plan). When the robot reaches point 9, the agent detects the ‘Tr’ intersection and recognizes that this intersection is represented by node 1 in the agent’s topological map. Turn left, go straight, and turn around operators are proposed. As always, the turn around operator is proposed with a worst preference. The agent knows that node 1 is connected to the start location. Since the object of interest has not been located, choosing the go straight plan that would lead back to the start location is an undesirable option. Production rules are fired that prefer the turn left plan over

the go straight plan (since the turn left plan would lead to an unexplored region) and the turn around plan over the go straight plan (since the turn around plan avoids returning to the start location). This means that although the turn around operator is proposed with a worst preference, productions can create preferences stating that a specific operator is worse than an operator with a worst preference. After these preferences have been generated, the agent selects the operator to set the turn left plan (this plan is chosen over the turn around plan because of the turn around plan's worst preference).

After completing the turn left plan to enter the side hallway at the bottom of Figure 6.25, the agent detects the doll before detecting the dead end intersection at the end of the hall. The SIFT matches used to detect the doll are shown in Figure 6.27. After detecting the doll (at point 10), the agent selects the returnHome operator (instead of the search operator that has been in use) and sets a plan for the robot to turn around to begin returning to the start location. At point 11, the agent detects an 'X' intersection and sets a turn left plan. As the agent is performing this plan, the wall in front of the robot is sensed and the 'T' intersection corresponding to node 1 of the topological map is detected and the turn left plan triggered by the 'X' intersection is interrupted.

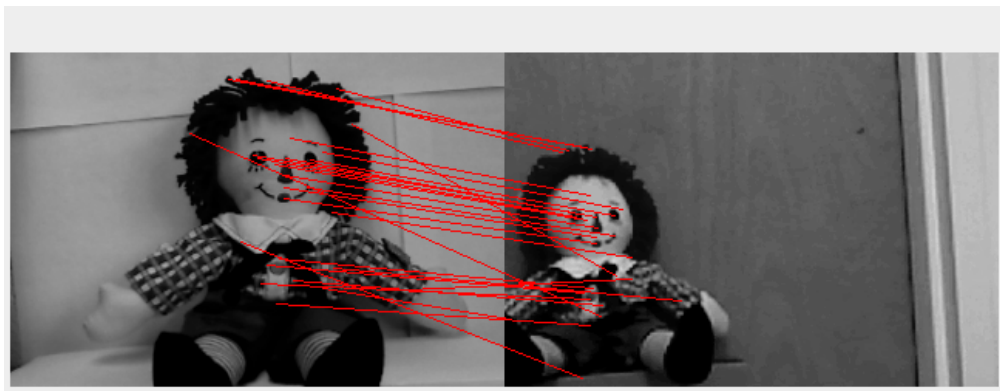


Figure 6.27: The SIFT matches between the training image and an image used to identify the doll during the test are shown.

When the agent is using the returnHome operator and detects an intersection that has previously visited, the operators for generating possible plans are proposed by considering the connections the current node has to its neighboring nodes in the agent's topological map. In this 'T' intersection, operators to generate turn

left and turn right plans were proposed. (Additionally, a turn around plan is proposed with a worst preference). Since more than one operator to set a plan (besides a turn around plan) is proposed, an operator tie impasse is generated. In the subgoal automatically generated to resolve this impasse, the agent uses Soar's planning mechanism to generate knowledge that Soar can use to select a plan that will move the robot closer to the test's start location. As described in Section 6.3.5.3, the agent internally simulates the effects of the candidate plans until it is able to resolve the impasse by creating an operator preference that allows Soar to choose between the tied operators. The operator to generate the left turn plan is chosen to be evaluated first. The simulation performed to evaluate this operator finds that choosing this plan will lead to the start location so this evaluation of this operator is success. This evaluation can be used to resolve the impasse and select the operator that sets the turn left plan. This result is saved as a chunk, that can be fired to generate this evaluation value if the agent needs to evaluate this operator in the future. This chunk is shown and discussed in Appendix C.

Once the turn left plan has been completed, the agent begins to monitor the distance the robot is from the middle of the 'T' intersection it has just moved through. When the robot is the same distance from this intersection as the length of the connection between the node in the topological map representing this intersection and the start location, the agent has considered the robot to have returned to the start location. The Soar agent then uses Soar's planning mechanism to generate instructions that can be used to get from the start location to the location the object of interest was found.

The instructions the agent generates at the end of the test from the start location to the object of interest are sent in an e-mail at the end of the test. The body of the e-mail is given here:

1. goStraight.
2. In T1, turnRight.
3. goStraight.
4. Look for doll.

This test demonstrated that the CRS was able to accomplish the task of searching an unknown indoor environment for an object of interest, while generating a

map of the environment that is useful for describing the location of the object. The agent was able to make useful decisions about the plans to perform at intersections it had visited before. Additionally, Soar's planning mechanism was used to make a decision about how to return the robot to the location at which the test started and to generate instructions to an object of interest that would be useful for a human. During this test, the agent was also able to interrupt plans generated as a result of incorrect intersection detections, preventing the topological map from being updated with inaccurate information.

6.4.2.2 Second test

For the second test in the same building, instead of selecting a plan to perform at random when the robot enters a new, previously unvisited, intersection, the Soar agent prefers turning plans over plans to go straight. In this test environment, this preference for selecting turning plans will encourage the agent to explore each side hallway the first time it is encountered. If the doll to be searched for is placed in the last side hallway, the agent's exploration of all side hallways will provide the agent with more options to consider when using Soar's planning mechanism to select plans to return the robot to the test's starting location.

Figure 6.28 shows the important events that occurred during the second test, as the CRS searched the same environment for the same doll as described in Section 6.4.2.1. The topological map generated by the agent during this second test is shown in Figure 6.29.

As in the first test, the robot started facing north at the location shown in Figure 6.28 by the green triangle. At point 1 (node 1 of the topological map), the agent selected turning right over going straight because of the modified preference for selecting turning plans. After detecting the dead end at point 2 (node 2 of the topological map), the robot turns around and returns to the first intersection it detected at point 3 (node 1). The agent chooses to turn right in this 'T' intersection to move to an area it has not visited yet. At point 4 (node 3), the 'TL' intersection is detected and the operator for creating a turn right plan is selected over the operator for generating a go straight plan again because of the modified Soar agent's preference for turn plans. After completing the turn right plan, the agent detects another dead end at point 5 (node 4). The agent then turns around and

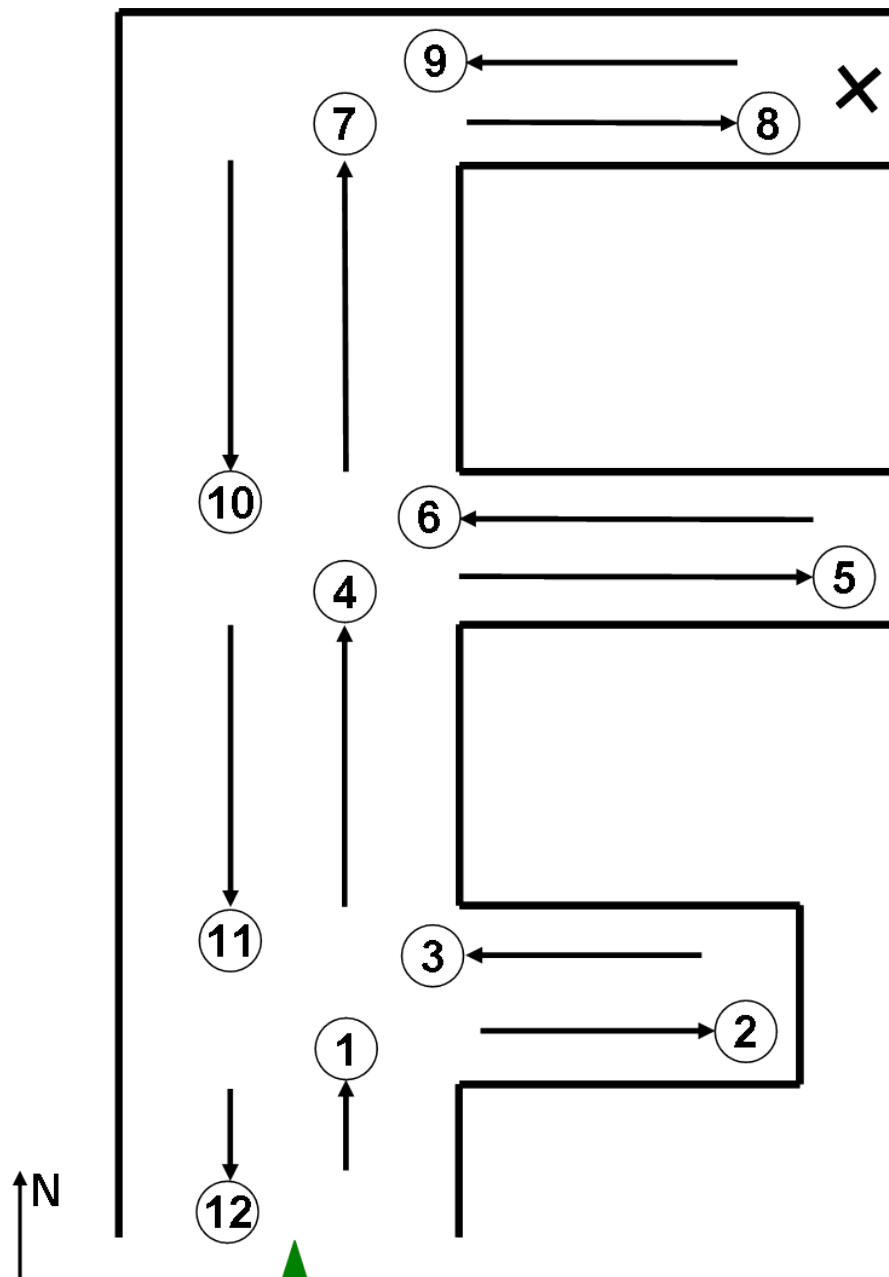


Figure 6.28: A map of the hallways used and arrows, labeled with numbers, indicating the order of movements the robot made through the environment during the test described in Section 6.4.2.2. The robot started at the green triangle and found the object of interest at the black 'X.'

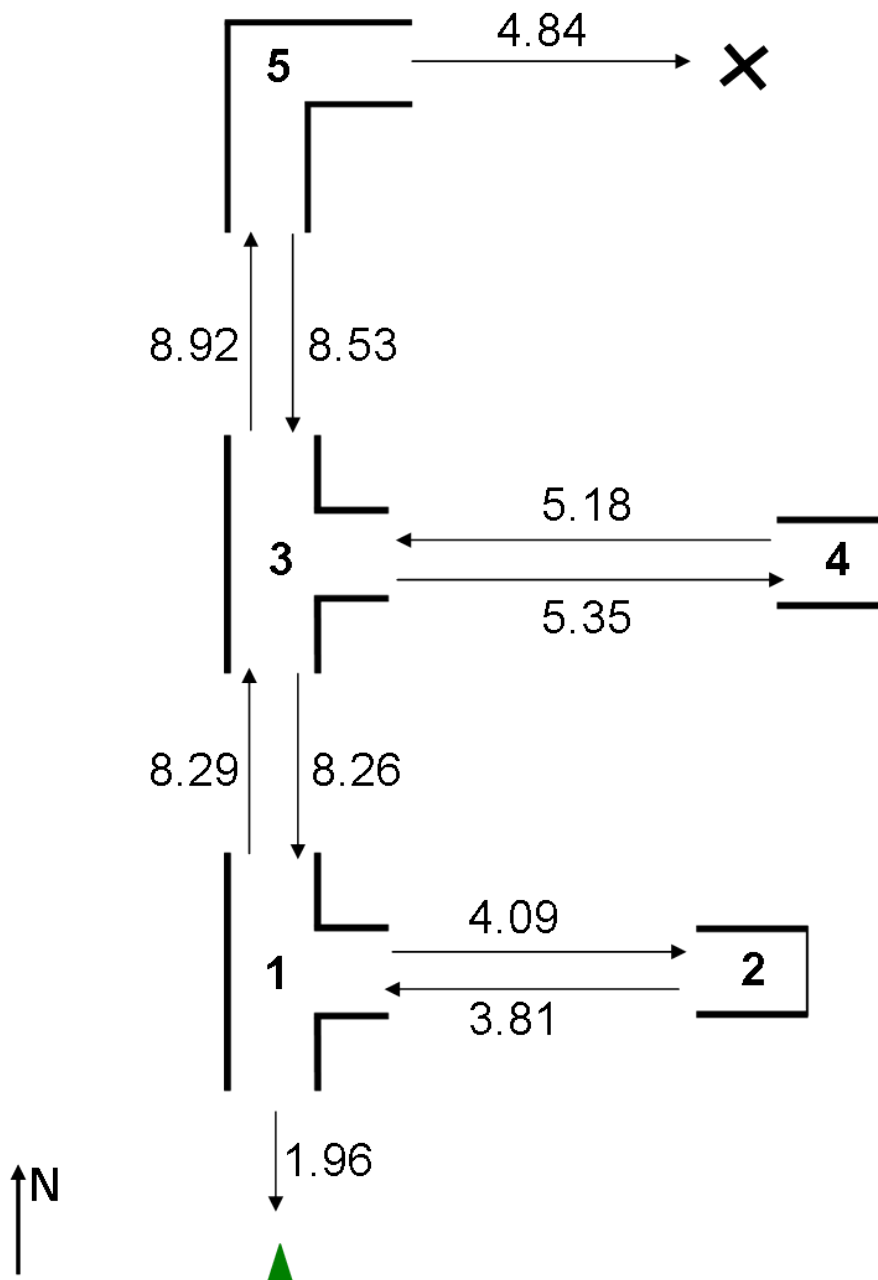


Figure 6.29: A depiction of the topological map generated by the agent during the test described in Section 6.4.2.2. The bold numbers 1–5 are the identification numbers given to each node in the map. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'

returns to node 3, where it detects a ‘T’ intersection at point 6. At this intersection, the agent selects a turn right plan to search a previously unexplored part of the environment. The robot moves north (up the figure) to point 7, where the agent detects an ‘R’ intersection (node 5). The robot moves east down the side hallway and detects the doll at point 8 before the dead end at the end of the hall is sensed.

After the doll has been detected, the agent turns around to return home. At point 9, the agent detects the ‘L’ intersection and generates a turn left plan. Shortly after finishing the turn left plan, the agent mistakenly detects a ‘Tr’ intersection while the robot is in a hallway (‘S’). Even though the ‘Tr’ intersection is incorrectly detected, the agent assumes this intersection is the intersection that corresponds to node 3 of the topological map and has been detected previously at points 4 and 6. Since the agent is returning home and there are connections from the current node (3) to nodes 4 and 1, the agent proposes operators to generate turn left and go straight plans. An operator tie impasse is generated because the agent does not know how to select between these operators. In the subgoal to resolve this impasse, the turn left operator is evaluated first. Since turning left would lead to a dead end, this operator is evaluated to be a failure. This evaluation gives the Soar agent enough information to select the operator to generate the go straight plan, resolving the impasse. Also, a chunk that stores the result of the evaluation of the turn left plan is learned. While the agent was performing this plan, it was able to correctly detect the ‘S’ hallway and use that detection to interrupt the go straight plan triggered in the incorrectly detected ‘Tr’ intersection. Although the planning was performed while the agent was not in the intersection it thought it was, the chunk learned in this situation has useful information and could be used in the future.

Once the go straight plan is interrupted, the navigate operator is used to move the robot toward node 3. At point 10, the agent correctly detects the ‘Tr’ intersection associated with node 3 of the topological map and uses Soar’s planning mechanism to choose between setting a turn left or go straight plan. Appendix D contains the trace generated by the Soar agent as it uses Soar’s planning mechanism to select the go straight plan.

First, the effect of the go straight plan is evaluated. The go straight plan is simulated and found to move the robot to node 1 of the topological map. The

agent is not able to complete its evaluation of the go straight operator at this point, so the agent considers the plans that are relevant at this ‘Tr’ intersection: turn left and go straight. Again, the agent is not sure what plan to choose, which generates a second operator tie impasse. The turn left operator is evaluated and found to lead to a dead end. This result is used to resolve the second operator tie impasse (the agent has enough knowledge to select the go straight plan) and is saved as a chunk.

The agent then simulates applying the go straight operator at node 1; the simulation results in the simulated robot reaching the start location. This result means that generating a go straight plan while facing south in node 3 can lead to the start location. This successful evaluation of the go straight operator is also saved as a chunk. Therefore, the agent selects the operator to generate a go straight plan.

At point 11, the ‘Tr’ intersection is detected and recognized as node 1; operators to set turn left and go straight plans are proposed. Again, an operator tie impasse results and Soar’s planning mechanism is used to attempt to generate the knowledge needed to select an operator. First, turning left is evaluated by simulating the effect of that plan. Since this plan would lead to the dead end detected at point 2 of this test, its evaluation is failure. This evaluation gives the agent enough knowledge to select the operator that sets a go straight plan. Again, a chunk is learned that stores the evaluation of a turn left plan when facing south at node 1 as failure. This chunk has some slightly different conditions for proposal than the chunk learned at point 10.

The agent performs this plan and as in the first test, uses the distance the robot has traveled from the last ‘Tr’ intersection to estimate when the robot has returned to the start location of the test. When the robot reaches the start location, Soar’s planning mechanism is used to generate the instructions from the start of the test to the doll given here:

1. goStraight.
2. In Tl, goStraight.
3. In Tl, goStraight.
4. In R, turnRight.
5. goStraight.

6. Look for doll.

This second test provided more opportunities for planning and learning than the first test, as the agent had explored the environment thoroughly and located the doll at a point in the environment where the agent could take advantage of the knowledge the agent learned about the environment while it was searching for the doll.

6.4.3 Test with loop

This section will describe a test performed in a environment different than the building hallways that were used for the tests described in Sections 6.4.2.1 and 6.4.2.2. Work benches in a lab classroom were used with cardboard pads to create a test environment with features, including a loop and an ‘X’ intersection, that were not present in the environment used for the first two tests. Figure 6.30 shows a picture of the SuperDroid in this environment.



Figure 6.30: A picture of the robot in the environment used for the test described in Section 6.4.3.

While the environment contains the same types of intersections the CRS was developed to detect, the Soar agent was not developed to consider the possibility of loops in the environment. The Soar agent used for this test selected plans to perform randomly at previously unvisited intersections, similar to the agent

used for the test described in Section 6.4.2.1. Other than a few small changes in parameter values, this agent is the same as the one used in Section 6.4.2.1.

The robot's movements through this environment are shown in Figure 6.31 and the topological map generated by the agent during the test is shown in Figure 6.32. The robot starts at the location of the green triangle in Figure 6.31 and moves north to point 1, where the agent detects a 'Tr' intersection and sets a plan to go straight. The robot continues to points 2, 3, and 4, detecting the 'L', 'X', and 'Tr' intersections at these intersections. When the robot reached point 5, a 'T' intersection is detected but the agent does not consider that this 'T' intersection belongs to the same node (node 1, as shown in Figure 6.32) as the 'Tr' intersection that was detected at point one. This means that after the turn left plan the agent selects to perform at point 5 is completed, the agent adds a new node (node 5) to its topological map. Not being able to recognize that the 'T' intersection at point 5 belongs to node 1 means that when the robot arrives at point 6, the agent is unable to recognize that it has previously visited this intersection before. So after completing the turn left plan set at point 6, the agent adds another new node (node 6) to its topological map. The same thing happens at point 7: the agent can not recognize that it has been at this intersection before, so after completing a go straight plan to move through the 'X' intersection, a new node (node number 7) is added to the agent's topological map.

At point 8, the agent detects both the doll and the dead end in which the doll is located. The agent sets a plan to turn around to begin to return to its start location; when this turn plan is completed, a new node for the dead end is added to the agent's topological map. At point 9, the agent detects the 'X' intersection and is able to recognize it as the 'X' intersection in node 7. When the agent reaches point 9, node 7 has a connection only to node 6. Therefore, the agent only proposes an operator to create a go straight plan that would move to robot to this node and this operator is selected. Since the agent knows it can follow the connections in its topological map back the the test's start location, it does not propose creating navigation plans that would cause it to visit unexplored regions of its environment. At point 10, the agent detects a 'R' intersection, associates the intersection with node 6, and sets a plan of turning right.

When the robot reaches point 11, the agent recognizes that it is at node 5 but

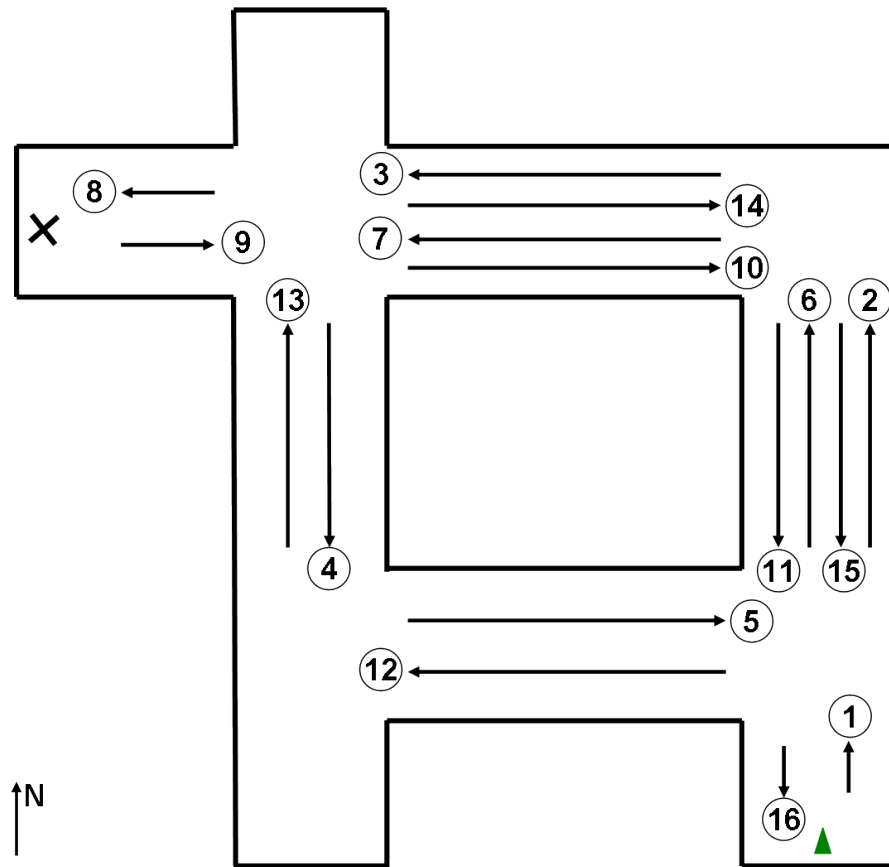


Figure 6.31: A map of the environment used for the test described in Section 6.4.3. The numbered arrows represent the order of the robot’s movements during the test.

does not know that node 5 is in the same location as node 1. Therefore, a plan is sent to follow the connection on the topological map to node 4. The robot moves around the loop through points 12, 13, and 14 (corresponding to nodes 4, 3, and 2, respectively). The robot arrives at point 15, detects the ‘Tl’ intersection associated with node 1, and sets a go straight plan. After completing this go straight plan and navigating towards the start location for the same distance as the the length of the connection between the start location and node 1 on the topological map, the robot stops at the start location to end the test. The directions the Soar agent generates to the doll’s location reflect that the agent is unable to recognize the presence of a loop in its environment:

1. goStraight.

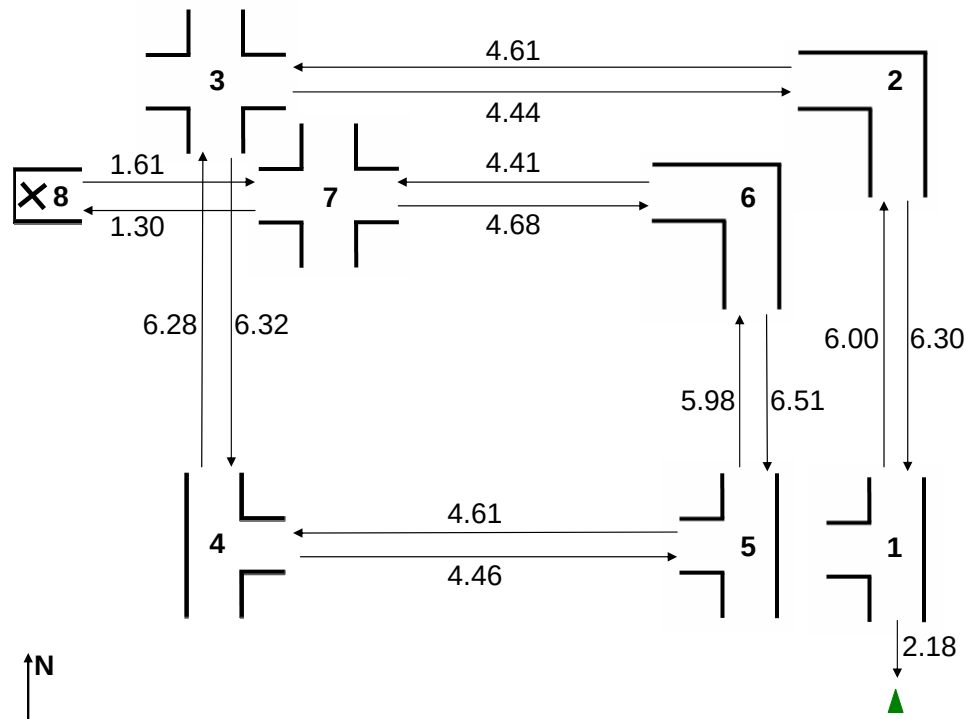


Figure 6.32: A depiction of the topological map generated by the agent during the test described in Section 6.4.3. The bold numbers 1–8 are the identification numbers given to each node in the map. The length of each connection in meters is given by the number next to the arrow for that connection. The robot started at the green triangle and found the object of interest at the black 'X.'

2. In Tr, goStraight.
3. In L, turnLeft.
4. In X, turnLeft.
5. In Tr, turnLeft.
6. In T, turnLeft.
7. In L, turnLeft.
8. In X, goStraight.
9. In D, look for doll.

While this test demonstrated that the Soar agent can function in an environment with a loop, not being able to recognize the presence of a loop increased the number of intersections the CRS needed to identify and the amount of time

needed to complete the test. While the current Soar agent does not consider the possibility of loops in the environment, it does have information about the environment in its working memory that could be used to consider this possibility. For this test, the agent would have needed to recognize it was returning to node 1 at point 5 of Figure 6.31 (when the agent was setting a plan to move through the ‘T’ intersection). The first possible source of information for determining this node has been visited before is to compare the estimated x and y position of the middle of the intersection detected at point 1 with the robot’s current position. These x and y coordinates are calculated from the odometry equations given in Section 4.1. The middle of the intersection detected at point 1 in Figure 6.31 was estimated to be (1.9, 3.92). The robot’s coordinate when the ‘T’ intersection was recorded at point 5 was (0.65, 4.23), which is 1.29 m away from the middle of the intersection detected at point 1. While this is a comparison of the middle of the first intersection to the first detection of the ‘T’ intersection, an estimate of the middle of the ‘T’ intersection is not available at this point (when a decision would have to be made as to whether the robot has been at this intersection before).

The connection lengths between nodes on the topological map could also be used to determine if the agent has visited this intersection before, indicating the presence of a loop. Using these lengths and calculating the distance traveled by the agent from node 4 to the location where the ‘T’ intersection is first recorded, the agent would have estimated that the robot was 0.54 m away from the intersection recorded at point 1 of the test. This decrease in distance from 1.29 m using raw odometry data to 0.54 m indicates the potential usefulness in the Soar agent using information from the topological map to recognize loops in an environment.

This test also demonstrated the utility of using topological information to reduce dependence on position estimates obtained using odometry. By using the topology of the environment to recognize when the robot returned to node 1 of the topological map and then navigating back to the start location for a distance equal to the length of the connection between the location where the test was started and the first node on the topological map, the robot was able to stop the test within about 40 cm of where it started the test. However, the estimate of the robot’s x,y location when the test was completed was greater than 6.5 m different than the x,y location of the robot at the start of the test, even though the physical distance

was only about 40 cm apart.

6.4.4 Simulation in more complicated environment

The Soar agent described in this chapter was able to use its topological map and Soar’s planning mechanism to make decisions that helped the robot return to the test’s starting location. However, the environments in which these test were performed were relatively simple topologically. It was difficult to perform tests in large environments; it can be hard to find buildings with interesting topological structures and even if such a building is found, tests in large environments can be time consuming. To demonstrate the usefulness of Soar’s planning mechanism for more challenging environments and to show in more detail how Soar’s planning mechanism can be used, an internal simulation was performed.

This simulation avoids the challenging robotics aspects of the tests described in Sections 6.4.2.1, 6.4.2.2, and 6.4.3. Instead of using sensor information and software algorithms to detect intersections, the agent in this simulation automatically knows the type of intersection it is in. And instead of performing navigation plans and using the navigate operator to move through the environment, the agent has productions that automatically moves a simulated robot from intersection to intersection. Except for these differences, the Soar rules used to take advantage of Soar’s planning mechanism in this simulation are the same rules used to set plans in previously visited intersections in the returnHome subgoal of the agent used for the tests described in Sections 6.4.2.1, 6.4.2.2, and 6.4.3. As in the rules used for planning by the Soar agent used to control the SuperDroid, the robot state in the simulation is represented only by the number of the robot’s current node and the robot’s current compass direction (north, south, east, or west).

The simulation assumes that the agent has already searched the environment, built a topological map, found the object of interest, and needs to decide how to return the location at which the test started. For this simulation, a description of the topological map shown in Figure 6.33 is placed in the Soar agent’s working memory. Node 0 represents the location where the test started and where the simulated robot needs to return to complete the test. To generate this topological map, the robot could have visited these nodes in the following order using the

search operator: 1, 2, 3, 4, 5, 6, 5, 1, 3, 4, 7, 8, 9, 8, 7, 10, 11, and then 12. The generation of this topological map assumes that loops in the environment can be recognized and incorporated into the topological map. The simulation assumes the object of interest was detected in node 12, while the simulated robot facing east (so that it is in a dead end). The part of the environment north of node 10 (represented by a dashed line in Figure 6.33) was not explored by the robot before finding the objects of interest so the agent is not aware of the nodes in that area of the environment and that area will not be considered as a path to travel to node 0 during planning.

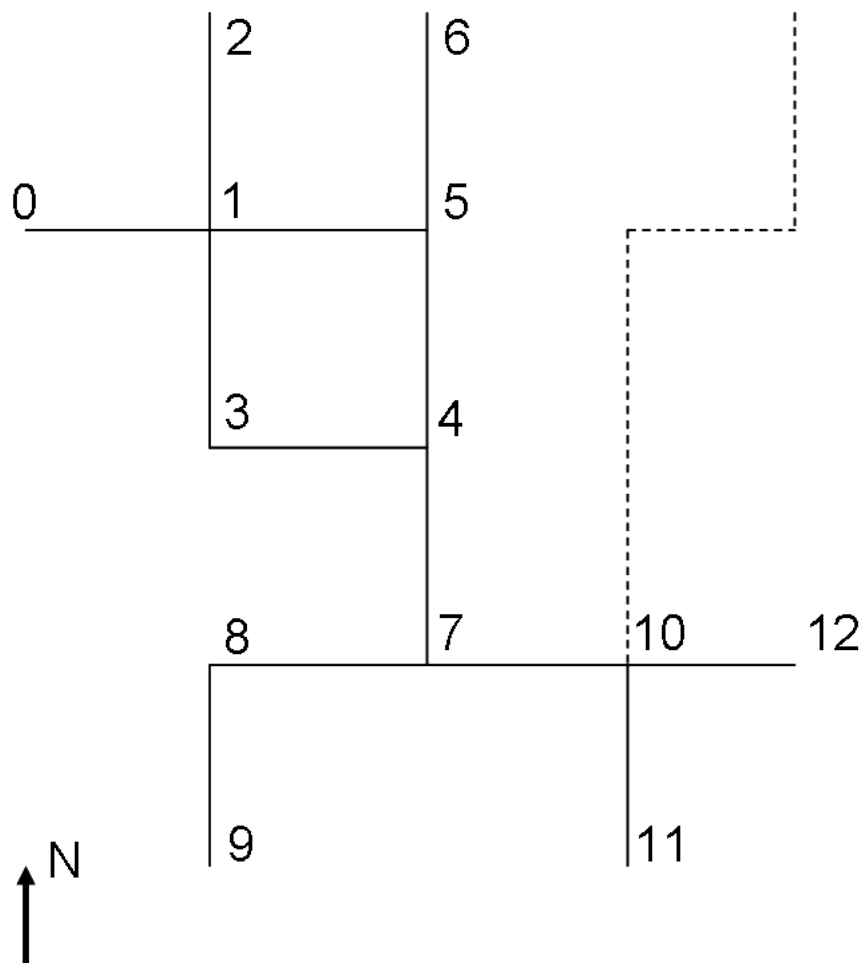


Figure 6.33: The topological map used for the simulations described in Section 6.4.4. The part of the map represented by a dashed line was not explored by the simulated robot.

Figures 6.34–6.39 show the progress of the agent as it moves from node 12 to node 0 to complete the test. In these figures, the level of subgoal is represented by the vertical axis and time is represented by the horizontal axis. When the agent is not able to make a decision about an action to take, an operator tie impasse occurs in which one of the tied operators is chosen for evaluation. It is possible to have an operator tie impasse while trying to resolve the initial operator tie impasse; each additional operator tie impasse is represented by moving down a level on the vertical axis. Each additional operator tie impasse also results in another copy of the simulated robot's state being created; the simulated operators in each level are performed on the copy of the simulated robot's state for that level. The goal location (node 0) on each individual map in Figures 6.34–6.39 is indicated by a green circle. The location of the simulated robot on each map is represented by the location of the yellow triangle. A planning failure occurs when the simulated robot (or a copy of its state) enters a dead end or reaches a duplicate state (that state is already contained in a higher level in the simulation) and is shown on a map as a red stop sign. Numbers and letters of the same type (e.g. capital letters or Roman numerals) refer to operators simulated at the same level. A dashed arrow pointing down and to the right represents creation of a subgoal to resolve an operator tie impasse; a dashed arrow pointing up and to the right means an impasse has been resolved and the agent has the knowledge to select one of the tied candidate operators; and a dashed arrow pointing to the right, which occurs only at the top level of these figures, represents moving to a new map to show the movement of the simulated robot. The numbers in the top right corners of each map are labels that will be used to refer to the individual maps in Figures 6.34–6.39. In the following description, moving to a bullet to the right of the current bullet indicates an operator tie impasse that results in the generation of a subgoal (moving down a level in the Figures 6.34–6.39).

- The simulated robot is initially facing east and in a dead end (node 12), as shown in the map 1 in Figure 6.34. The only operator that is proposed by the agent in this situation is the turn around operator, so it is selected and applied.
- The robot turns around and enters node 10 (an 'X' intersection) facing west

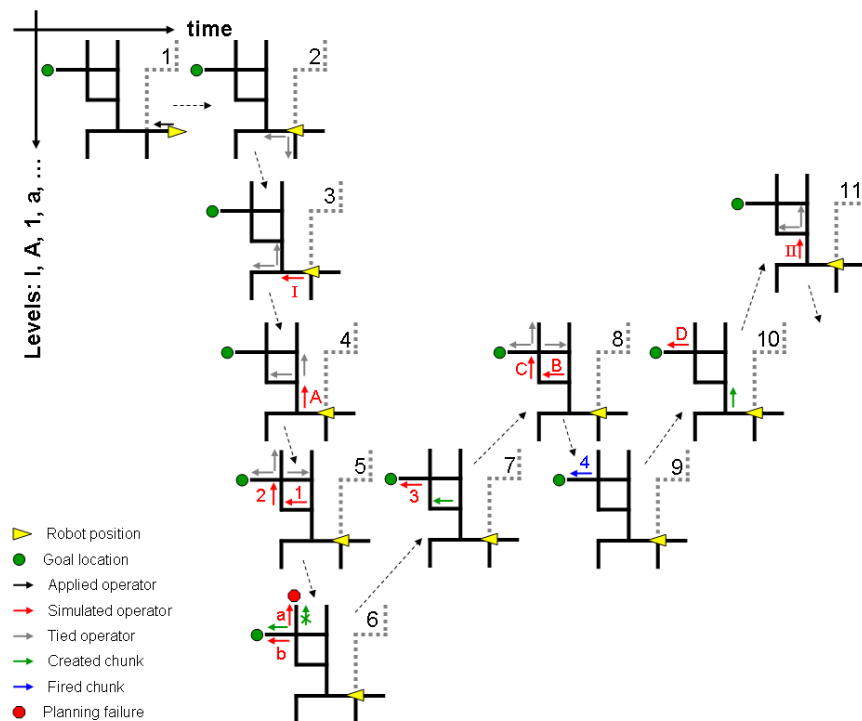


Figure 6.34: Figure 1 of 6 showing planning in simulated test (continued in Figure 6.35).

(shown in the map 2 of Figure 6.34). Here, there are several operators proposed: go straight, turn left, and turn around. Although there is an area of the environment to the north of node 10, the agent does not know any information about this area (indicating that it has not explored) so a turn right operator is not proposed. Also, throughout this simulation, the turn around operator is proposed with a worst preference (meaning it will not be selected unless there are no other operators proposed or other proposed operators are found to be less desirable than turning around). Even though the turn around operator is always proposed, this proposal will not be mentioned for the rest of the description of this simulation as there are always other proposed operators in this simulation. So while the agent knows that the go straight and turn left operators are more preferable than the turn around operator, it is unable to choose between the go straight and turn left operators, resulting in a operator tie impasse with the two tied operators indicated by the two

gray arrows in map 2.

- In the subgoal to resolve the operator tie impasse, the go straight operator (I) is evaluated first (map 3). After the effect of the go straight operator is simulated (moving a copy of the simulated robot to node 7, facing west), the agent proposes the go straight and turn right operators, generating another operator tie impasse.
 - The turn right operator (A) is evaluated first, moving a copy of the simulated robot to node 4 (facing north), as shown in map 4. Here, another operator tie impasse occurs after the turn left and go straight operators are proposed.
 - * The turn left operator (1) is evaluated, causing a copy of the simulated robot to move to node 3, while facing west (map 5). Here the agent proposes, selects, and applies the turn right operator, which moves the copy of simulated robot to node 1 (facing north). At this point, turn left, go straight, and turn right operators are proposed and an operator tie impasse is generated for these three candidate operators.
 - First, the go straight operator (a) is evaluated (map 6). This operator moves the copy of the simulated robot into a dead end, which is a state of failure. Therefore, the evaluation of applying the go straight operator while facing north in node 1 is failure. This result is saved as a chunk. The agent is still not able to create a preference to choose between the turn left and turn right operators, so the agent remains in the subgoal to resolve the impasse. The turn left operator is evaluated next; this operator leads to the goal state. The evaluation of the turn left operator when the simulated robot is facing north in node 1 is success. Again, a chunk is created that stores this result.
 - * The agent uses its knowledge of the successful evaluation of the turn left operator when the agent is facing north in node 1 and selects this turn left operator to apply (map 7). As expected, this operator moves the copy of the simulated robot to the goal location, confirming that

the turn left operator proposed after the application of simulated operator A can lead to a successful result. A chunk is created that stores the result that if the robot is facing north in node 4, the evaluation of a turn left operator is success.

- Now the agent uses the result that if the robot is facing north in node 4, the turn left operator can lead to success to apply the turn left operator (operator B in map 8). When the copy of the simulated robot is facing west in node 3, it proposes the turn right operator. The agent is able to select and apply this operator since there are no other candidate operators without a worst preference. When this operator moves the copy of the simulated robot to node 1 (facing north), the agent proposes the turn left, go straight, and turn right operators. All three candidate operators have the same preferences, resulting in an operator tie impasse.
 - * The agent chooses to evaluate the turn left operator first (map 9). However, the agent has already evaluated this operator (simulated operator b from map 6) and built a chunk to save the successful evaluation. Therefore, the agent is able to fire this chunk and generate a successful evaluation of this operator without having to simulate the operator.
- The successful evaluation of the turn left operator is used to select this operator. It is applied (operator D in map 10), leading to the copy of the simulated robot at this level reaching the goal location. This result (the evaluation of a turn right operator when the robot is facing west in node 7 is success) is saved in a chunk.
- The successful evaluation of the turn right operator when the robot is facing west in node 7 is used to resolve the operator tie impasses depicted in the map where operator I was applied (map 3). The turn right operator is selected over the go straight operator and applied (operator II). When the copy of the simulated robot for this level (shown in map 11) enters node 4 while facing north, an operator tie impasse occurs to create knowledge to choose between the tied go straight and turn left operators.

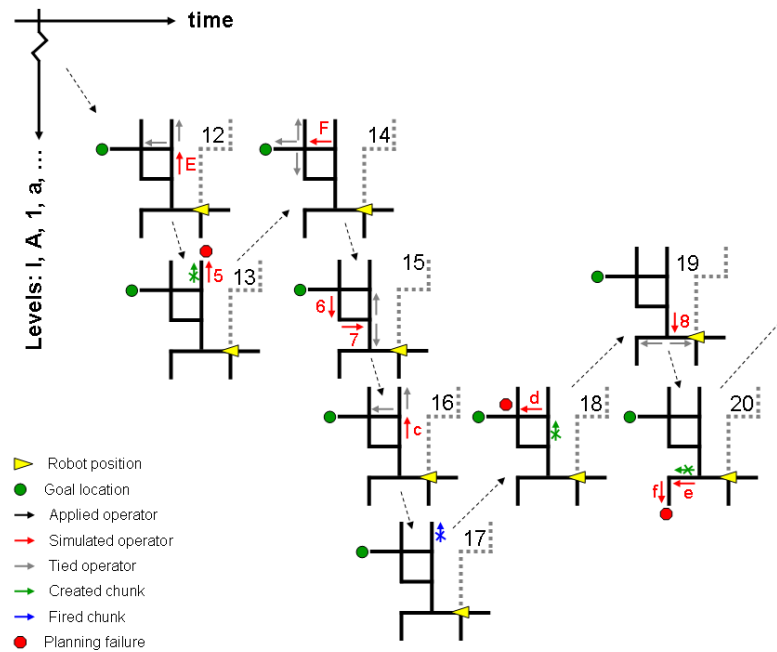


Figure 6.35: Figure 2 of 6 showing planning in simulated test (continued in Figure 6.36).

- In the subgoal to resolve this impasse (map 12 in the upper left of Figure 6.35), the go straight operator is chosen to be evaluated first. The vertical axis on this figure has an axis break, as there are no maps depicting top two levels shown in this figure. The simulated operator (E) applied in map 12 moves the copy of the simulated robot to node 5 (facing north). Here, turn left and go straight operators are proposed, resulting in another operator tie impasse.
 - * The go straight operator is again chosen to be evaluated first (map 13). The simulation of this simulated operator (5) leads to a planning failure when the copy of the simulated robot enters node 6 (a dead end). The failure of the go straight operator in this situation is saved in a chunk.
- The evaluation of the go straight operator as a failure generates enough knowledge for the agent to choose the turn left operator. This operator (F) is applied (map 14), moving the copy of the simulated robot at this level to node 1 (facing west). The turn left, go straight, and turn right

operators are proposed and result in another operator tie impasse.

- * The turn left operator is chosen for evaluation first. It (operator 6 in map 15) moves the copy of the simulated robot to node 3 (facing south), where the left operator is proposed, selected, and applied (operator 7). At node 4, the turn left and turn right operators are proposed, but the agent is unable to choose between them, leading to an operator tie impasse.
 - The turn left operator is evaluated (operator c in map 16), moving the copy of the simulated robot to node 5 and resulting in another operator tie impasse between the turn left and go straight operators.
 - ◊ The go straight operator is chosen for evaluation. This operator had already been evaluated in this situation (simulated operator 5 in map 13), so the corresponding chunk (shown in map 17) was fired to indicate that the application of this operator would lead to a failure without the need to simulate the operator.
 - The agent used the evaluation of the go straight operator as a failure to select and apply the turn left operator (d, in map 18). However, this also leads to a failure as the Soar agent recognizes that the state of the copy of the simulated robot at this level (facing west in node 1) is also present at a higher level (after the application of operator F in map 14). This means that the selection and application of operator c can only lead to states of failure (and the evaluation of operator c as a failure is stored as a chunk).
- * The evaluation of the tied turn left operator in map 15 as a failure results in the agent applying the turn right operator (operator 8 in map 19). After the copy of the simulated robot enters node 7 facing south, another operator tie impasse (between turn left and turn right operators) occurs.
 - The turn right operator (e) is evaluated first (map 20). After

the turn left operator (f) is applied in node 8, a failure state is encountered in node 9 (a dead end). A chunk that stores that the evaluation of a turn right operator when facing south in node 7 as failure is learned.

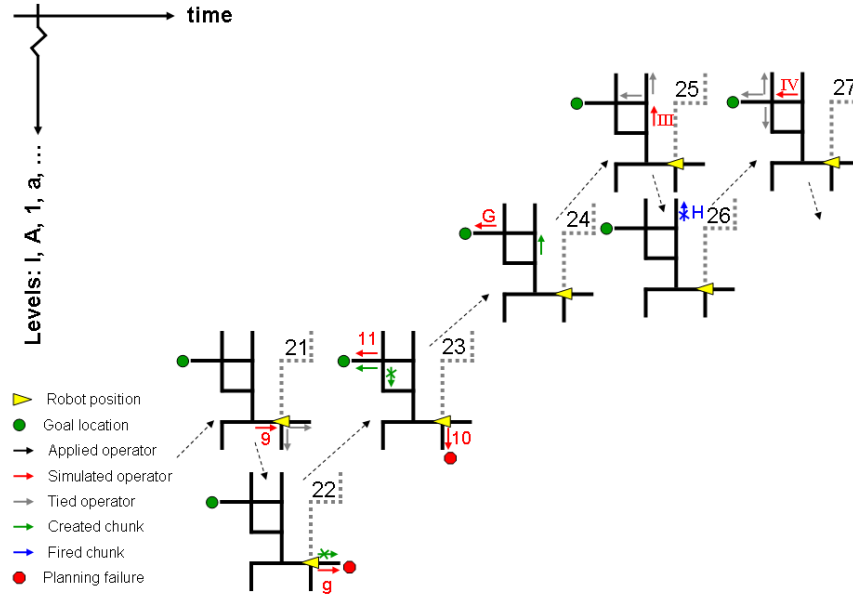


Figure 6.36: Figure 3 of 6 showing planning in simulated test (continued in Figure 6.37).

- * The evaluation of the tied turn right operator from map 19 as failure results in the selection and application of the turn left operator (operator 9 in map 21 shown in Figure 6.36). After the turn left operator is used to move the copy of the simulated robot to node 10, the agent proposes go straight and turn right operators. An operator tie impasse results.
- The go straight operator (operator g) is evaluated first; its application in map 22 results in a planning failure when the copy of the simulated robot enters a dead end. A chunk storing the evaluation of failure for the go straight operator when the robot is facing west in node 10 is created.

- * Since the turn right operator tied in map 21 was evaluated to lead to a failure state, the go straight operator is selected and applied (operator 10 in map 23). The application of this operator also leads to a failure state (the dead end of node 11). This means that all possible plans resulting from the left turn simulated by operator 6 (from map 15) lead to failure states. This result is stored a chunk that the evaluation of a turn left operator when facing west in node 1 is failure. All of the internally simulated operators shown in maps 15–23 have been to evaluate the left turn operator from the operator tie impasse between the three operators shown in map 14. The turn left operator has been evaluated to be a failure, but the tie between the go straight and turn right still exists. The go straight operator (11) is evaluated next; it is successful and this result is stored as a chunk.
 - Using the evaluation of the go straight operator (11) as a success, the copy of the simulated robot in the next level up (shown in map 24) is moved to the goal location using operator G. This means that the go straight operator at node 4 when facing north (simulated by operator E in map 12) can lead to the goal location. A chunk using this result is learned.
- Using that the evaluation of operator E was found to be success, the go straight operator, whose original proposal was shown in map 11, is applied in map 25 (operator III). This operator application moves the copy of the simulated robot depicted in map 25 to node 5 (facing north). Here, the agent proposes the turn left and go straight operators, which results in another operator tie impasse.
 - The go straight operator from the operator tie impasse in map 25 is evaluated first. There is a chunk that has stored the evaluation of the go straight operator in this situation that is fired (H) instead of simulating the operator.
- Using the evaluation of the go straight operator as failure, the turn left operator is selected and applied (operator IV). Map 27 then shows the

resulting operator tie impasse between the turn left, go straight, and turn right operators proposed when the copy of the simulated robot is facing west in node 1.

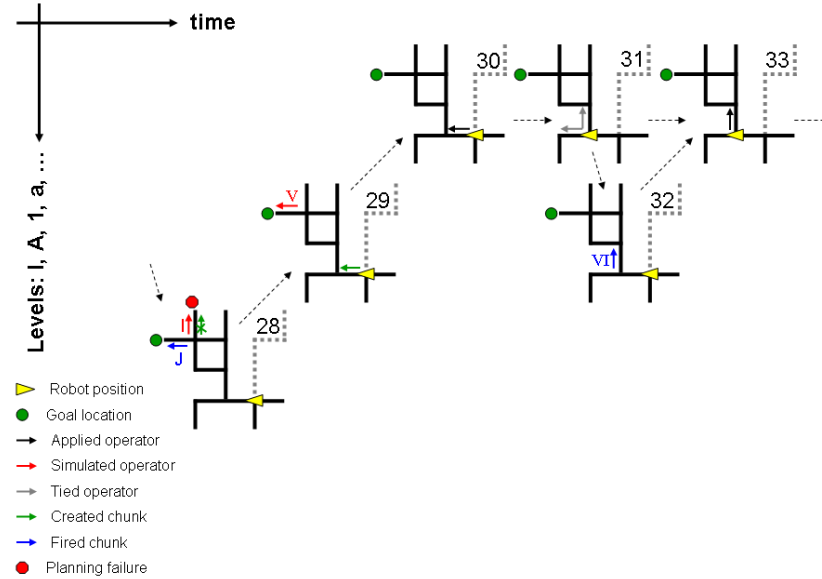


Figure 6.37: Figure 4 of 6 showing planning in simulated test (continued in Figure 6.38).

- The turn right operator (I in map 28) is evaluated first: it leads to the dead end in node 2 and therefore a failure state. A chunk is generated to store the result of this evaluation. The go straight operator is then selected for evaluation (before the turn left operator). A previously learned chunk is used to evaluate the go straight operator as successful (J).
- The agent uses the successful evaluation of the go straight operator to select and apply the go straight operator (V) shown in map 29. This result allows the evaluation of the go straight operator from the operator tie impasse generated in map 2 to be completed and found to be successful. A chunk is created that says that the evaluation of a go straight operator while facing west in node 10 is a success.

- The agent applies the go straight operator (in map 30) to move the simulated robot to node 7, which it enters facing west (shown in map 31). All of the simulated operators shown in maps 3–29 have been to create the knowledge needed to resolve the operator tie impasse depicted in map 2 and then select the go straight operator shown in map 30. In node 7, the agent proposes go straight and turn right operators, resulting in an operator tie impasse.
 - The turn right operator is evaluated first: a chunk fires that evaluates this operator as successful (VI).
- The turn right operator is applied (shown in map 33), moving the simulated robot to node 4 (shown in map 34, in Figure 6.38). While facing north in node 4, the turn left and go straight operators are proposed and an operator tie impasses results.

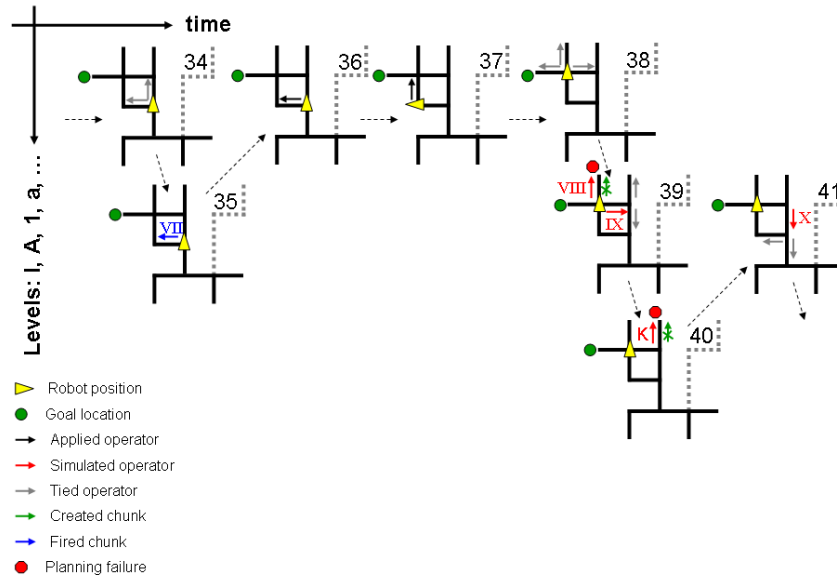


Figure 6.38: Figure 5 of 6 showing planning in simulated test (continued in Figure 6.39).

- The turn left operator is chosen for evaluation before the go straight operator. A chunk is used to evaluate the operator as successful (VII). Note

that if the go straight operator had been selected for evaluation first, the chunk shown in map 24 would have been used to evaluate the go straight operator as successful as well.

- The successful evaluation of the turn left operator allows the agent to resolve the impasse and select and apply this turn operator (shown in map 36). When the simulated robot enters node 3 facing west, it is able to propose, select, and apply the turn right operator (shown in map 37).
- The simulated robot enters node 1 facing north. The turn left, go straight, and turn right operators are proposed and generate an operator tie impasse. Map 38 shows these three tied operators.
 - The go straight operator is evaluated first (VIII, in map 39). It leads to a failure state (the dead end in node 2) and this result is saved as a chunk. The turn right operator is evaluated next (instead of the turn left operator). This simulated operator (IX) moves the copy of the simulated robot to node 5, which it enters facing east. The proposed turn left and turn right operators result in an operator tie impasse.
 - The turn left operator is evaluated first by simulated operator K (map 40). This operator leads to a dead end and a chunk that saves the evaluation of this operator as a failure is created.
 - Since the turn left operator was evaluated as a failure, the turn right operator is selected and applied (operator X). The copy of the simulated robot enters node 4 facing south and proposes the go straight and turn right operators. These tied operators generate an impasse, shown in map 41.
 - The turn right operator (L) is evaluated first (as shown on map 42 in Figure 6.39). In node 3, the turn right operator is used to move the copy of the simulated robot to node 1, which it enters facing north. This state of facing north in node 1 is the same state in which the simulated robot is currently located. This duplicate state means the agent has encountered a loop somewhere in its planning, which indicates

a planning failure. A chunk storing the result that a turn right operator chosen when entering node 4 facing south leads to a failure state is created.

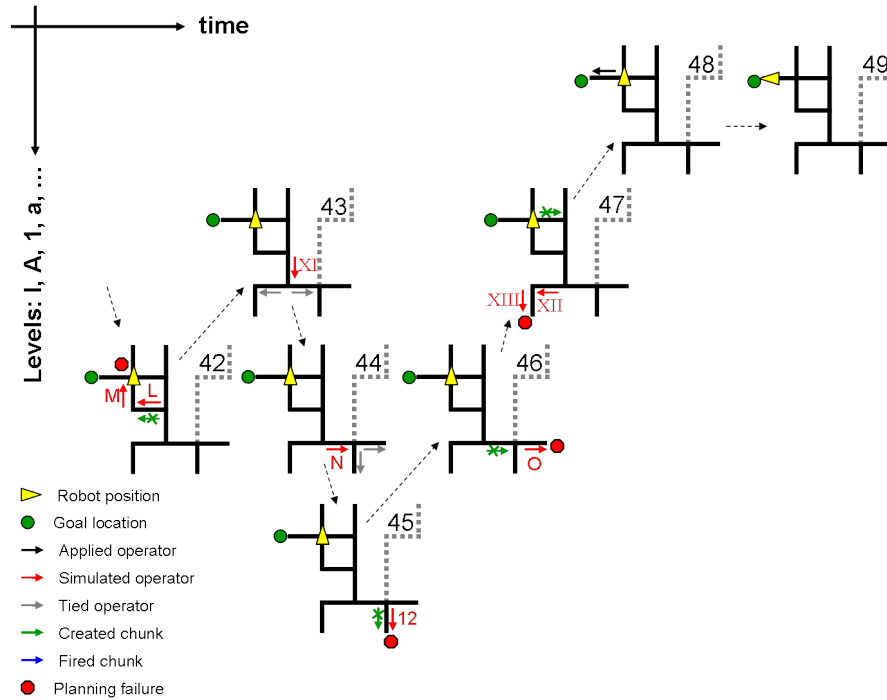


Figure 6.39: Figure 6 of 6 showing planning in simulated test.

- The evaluation of the turn right operator simulated by operator L allows the Soar agent to select and apply the go straight operator (XI on map 43). After operator XI is applied, the turn left and turn right operators are proposed as the copy of the simulated robot enters node 7 facing south. These two candidate operators have the same preferences and result in an operator tie impasse.
 - The turn left operator is chosen to be evaluated first. Its simulation (operator N) results in the state of the copy of the simulated robot being changed to facing east in node 10. Here, an operator tie impasse occurs between the tied go straight and turn right operators (map 44).
 - * The turn right operator is evaluated first and its simulation (operator 12) leads to a failure state (map 45). A chunk saves the result that

the evaluation of the turn right in node 10 while facing west is a failure.

- The go straight operator is selected and its simulation (operator O, shown in map 46) also leads to a failure state. This results in a chunk that saves the evaluation of a left turn operator in node 7 (facing south) as a failure.
- The simulations shown in maps 44–46 were used to evaluate the tied turn left operator shown in map 43. With that operator evaluated to be a failure, the tied turn right operator is selected and applied (operator XII in map 47). After operator XII is applied, operator XIII is applied in node 8 and leads to a dead end. The results of these simulated operators mean that all plans that come after the application of simulated turn right operator IX, which is shown in map 39, result in failure. A chunk storing this result is learned.
- Both the tied turn right and go straight operators shown on map 38 have been evaluated as failures. This gives the agent enough information to select the turn left operator for application (shown in map 48).
- Map 49 shows the simulated robot has successfully reached the goal location.

During the simulation described by Figures 6.34–6.39, 42 operators were internally simulated and 19 chunks were created. The agent was able to use six chunks sometime later in the simulation after they had been created. While the use of these six chunks saved the agent from internally simulating some operators, if another simulation is performed and the agent is allowed to remember the chunks created in the previous simulation, the amount of internal simulation the agent needs to perform can be dramatically reduced. To illustrate this, another simulation of the agent planning to move the simulated robot from node 12 to the goal location was performed. Figures 6.40 and 6.41 show the planning needed to perform this simulation. No internal simulations of operators needed to be performed: for every operator that was selected for evaluation, there was a corresponding chunk that had been learned in the previous simulation. This reduction in the number of internally simulated operators illustrates a transition from deliberative to more reactive behavior.

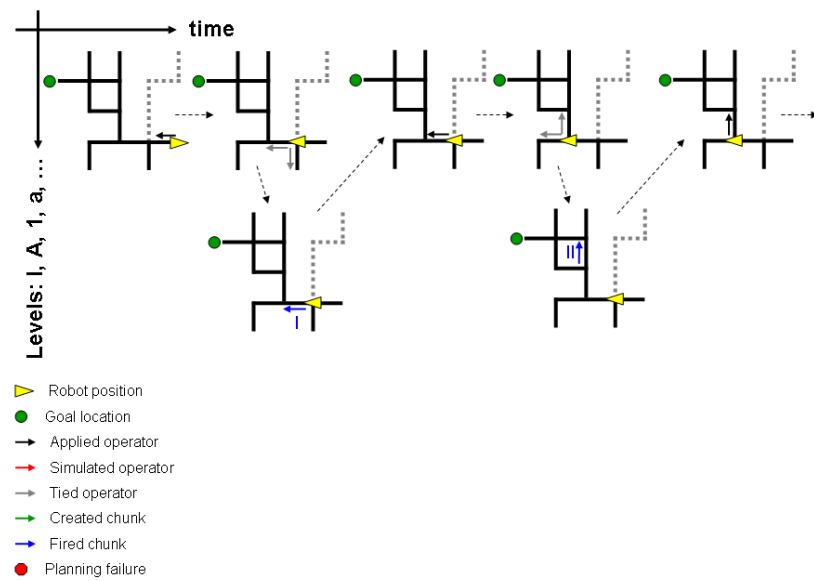


Figure 6.40: Figure 1 of 2 for the simulation performed with knowledge of chunks generated in the previous simulation described in Figures 6.34–6.39 (continued in Figure 6.41).

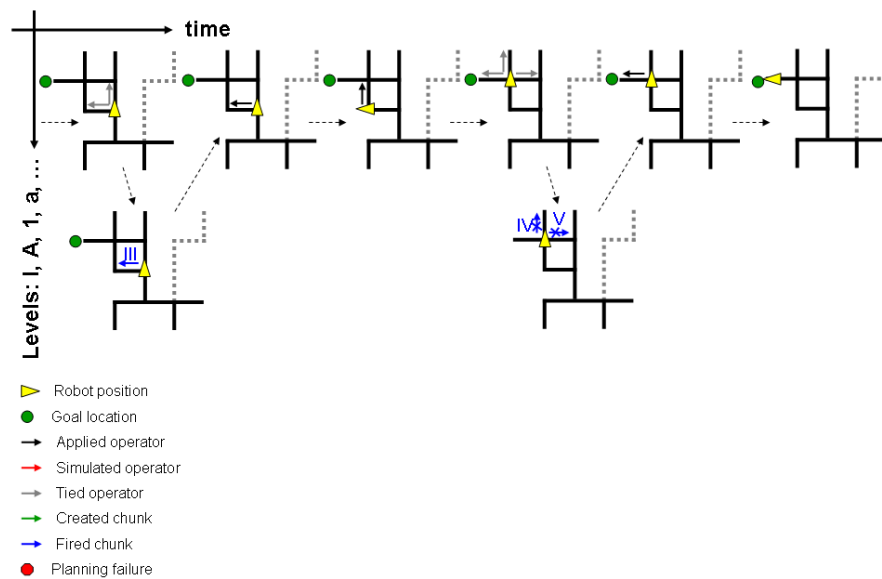


Figure 6.41: Figure 2 of 2 for the simulation performed with knowledge of chunks generated in the previous simulation described in Figures 6.34–6.39.

The results described in this section demonstrated the Soar agent using chunking to generate more reactive behavior both within a single simulation (the six chunks used in the first simulation) and in consecutive simulations. In the current Soar agent, the chunks created store the results of the evaluation of operators. This means that even if the agent is in a situation that it has been before, the previously learned chunk will not fire until the appropriate operator is selected for evaluation (since the evaluation of the operator is a condition of the chunk). However, changing the state in which learning is turned on can lead to the generation of chunks with different conditions. For example, chunks can be learned that give an operator a best preference as soon as it has been proposed, as opposed to an evaluation of success when it is selected for evaluation. This type of chunk may prevent agent from trying out the other candidate operators (if the first operator evaluated succeeds and a chunk is learned that fires and gives that operator a best preference whenever it is proposed, the other candidate operators for that situation may never get the chance to be evaluated).

6.5 Discussion

The system described in this chapter has successfully searched inside a building, found intersections that could be used as nodes in a topological map, looked for an object of interest, returned to its initial location using information from the topological map, and generated instructions to move from its initial location to the object of interest using intersections as landmarks. To accomplish these tasks, the CRS integrated the Soar architecture with traditional robotic mapping and computer vision algorithms, as well as other software systems (including motor control and fuzzy logic systems). The Soar agent was able to use meaningful symbols that referred to intersections and objects in the vehicle's environment to generate an internal model of the environment that was useful for taking advantage of Soar's planning and chunking mechanisms.

While the CRS was successful in showing the usefulness of Soar for the control of a mobile robot for the tests described in this chapter, there are a number of improvements that could be made to this system. There were three weaknesses identified during development of the CRS that arise from the limited sensing ca-

pabilities of the CRS.

The first is that the Soar agent can have trouble placing goal waypoints in desirable locations when it is not able to directly measure a feature of an intersection that can be used to calculate the goal location. For example, this can happen when the robot is navigating down a wide hall and comes to a narrow hall on its right (identified as a ‘Tl’ intersection) and the Soar agent decides to set a plan to turn right to enter the hall on the side of the robot. The agent does not have access to information it could use to determine the width of the opening into the side hall, so it uses a percentage of the width of the current hall to calculate the goal waypoint of the first goal of this turn right plan. Additionally, after completing the first two goals of the turn right plan, the third goal is used to enter the side hall. However, the agent does not have enough knowledge about the area in front of the robot to locate the walls in the new hall and may set a goal waypoint that is near one of the walls (or even behind one of the walls). As an alternative, the agent could use the techniques developed for the navigate complex operator instead of navigation plans to move through intersections. For example, the agent could follow a wall on its right side to execute a right turn into a new hall. Using this strategy would reduce the opportunity for the agent to confirm it had correctly identified the type of intersection that was used to set the plan (currently done during the first goal of a turn plan), which is very useful to the Soar agent given its current sensing capabilities.

The second weakness is that if the robot is close to a wall on one of its sides after having completed a turn navigation plan to move through an intersection, it can take more distance than is desirable for the agent to return the robot to the middle of the hall. Part of the reason this happens is that it is easier for the agent to move and stay in the middle of the hall if it has accurate information about the hall’s width and orientation from the Hough transform algorithm. However, after completing a turn navigation plan, there is no information about the walls in the new hall available in the occupancy grid for the Hough transform to identify. The robot needs to move forward into the new hall so the occupancy grid can reflect the presence of walls on the sides of the robot. Only then can the Hough transform algorithm identify these walls and the width and orientation of the current hall can be used for reasoning by the Soar agent.

The third weakness resulting from limited perception is the inability to localize small obstacles in front of the robot. Since only information from one sonar sensor is used to detect obstacles in front of the robot and that sensor has a wide perceptual field, the agent is able to determine if there is an obstacle in front of the robot but has little idea about the size of the obstacle or where the object is located within the sensor's perceptual field. These three weaknesses could be improved with the use of additional sensors. While more modern sensor systems such as a laser rangefinder or stereo cameras could be used to more accurately localize obstacles in front of the robot, even additional sonar sensors facing to the side could be used to estimate the orientation of robot relative to hall without the use of an occupancy grid [98]. Improved sensing capabilities could also allow the CRS to detect additional types of intersections (e.g. intersections that have entrances and exits in directions other than the four cardinal compass directions). It should also be noted that with the definition of the appropriate inverse sensor models, information from sensors such as laser rangefinder or stereo cameras could be incorporated into the occupancy grid algorithm.

There are also some limitations in how the Soar agent constructs its topological map. Currently, the Soar agent does not consider that the environment may contain loops or that incorrect information may have been added to the topological map. While the Soar agent attempts to avoid adding incorrect nodes to the topological map by waiting until the navigation plan through an intersection has been completed before adding that intersection to the topological map as a node, the potential to add incorrect nodes still exists. Another situation that the Soar agent has difficulty recovering from is the case where an intersection detection is missed. This may lead the agent adding an incorrect connection between two nodes on the map that are not actually connected; the agent has no way to realize an incorrect connection exists and no mechanism to attempt to fix this inaccurate connection. While in some cases the CRS is able to complete the indoor search mission even while the topological map contains incorrect information (e.g. the missing loop in the test described in Section 6.4.3), in others it becomes difficult for the agent to know when the robot has returned to the location where it started the test.

Future Soar agents could introduce some uncertainty about the structure of the topological map by maintaining a tree of potential topological maps or considering

the possibility that the agent may not be sure where the robot is located with respect to the topological map. The agent could use information such as connection lengths between nodes of the topological map (this information is currently included in the topological map) or use additional sensing capabilities to distinguish between intersections of the same type (e.g. using SIFT features as described by Se et al. [122] and Wang et al. [100]).

Another improvement that could be considered is using a popular method to reduce the error in estimating the robot state using odometry information (e.g. continuous localization [54] or particle filters [90], [91]). More accurate state estimates may provide helpful information to consider when determining the appropriate topological structure for an environment. There are also other changes in the Soar agent that could be beneficial as well: for example, implementing a more formal strategy for searching the environment for objects of interest and taking advantage of newer Soar features, such as episodic memory, semantic memory, or reinforcement learning [22].

There are aspects of the system described in this chapter that are similar to system developed by other researchers. Laird, Derbinsky, and Voigt [47] described a Soar agent that was able to control a simulated robot that looked for blocks in a simulated building and moved specific types of blocks to a specified storage room within that environment. The agent constructed a map of the building, which contained 25 rooms in a straight line connected by 24 doorways, that described the rooms and doorways and their spatial and topological layout and the location of the blocks it located. The agent uses internally simulated operators similar to those used by the Soar agent described in this chapter to search for a path to a specified room in the simulated building. While a real robot that could perform a subset of the sensing needed for the test performed in the simulated environment was described, the presented results were obtained using a simulated robot. The agent described could store map information about the environment in working, episodic, or semantic memory. So there are similarities between this Soar agent and the agent developed for the indoor search mission described in this chapter, but the Soar agent developed by Laird et al. took advantage of some of the new architectural mechanisms in Soar. However, the results generated by the agent developed by Laird et al. were obtained in a simulated environment and while it

is mentioned that the same agent could be used on the real robot, the type of information sensed in the simulated environment (such as the room the robot is in, the location of walls and doors, and the presence of other objects) could be more challenging to sense in a real environment using real sensors.

Other groups have also used maps with cognitive architectures. Kennedy et al. [23] integrated a mapping capability in a system with a cognitive architecture: their system used occupancy grids and a 2-D grid-based cognitive map. The cognitive map was maintained outside of ACT-R, but the ACT-R model could access the map for reasoning about relative locations of objects represented in the map. So while our system also used an occupancy grid, the type of information extracted from the maps for use by the cognitive architecture in the two systems is different as one system uses grid-based information while the other uses topological information. More recent work by this group has focused on tasks other than navigation such as conversation tracking and gaze following and has included some useful sensing capabilities such as tracking people and localizing the sources of sounds [49], [55]. These kinds of tasks seem to make better use of the strengths of ACT-R than an indoor search mission that includes the use of planning to reach a desired location.

Kelley et al. [26] have used SS-RICS to identify common intersection types and make navigation decisions based on knowledge about paths available in each type of intersection. There are several differences between the work described in this chapter and the work described by Kelley et al. [26]. First, in addition to navigating through an indoor environment while identifying intersections, the CRS is also able to look for an object of interest, record information about the connections between recorded intersection, and use these connections and Soar's planning mechanism to search for paths through the environment to a specified location. While the CRS uses the probabilistic framework of the occupancy grid to combine sensor data over time to detect intersections, the SS-RICS can take advantage of the more powerful sensing capabilities of a laser rangefinder to directly use sensor information for intersection detections. The SS-RICS used odometry information to recognize when it was seeing a previously visited intersections, while the CRS attempts to use information about the connections between intersections that has been stored in the Soar agent's topological map. The search strategies used by the two systems also are different: Kelley et al. used a maze solving algorithm to guide navigation

decisions in SS-RICS. The Soar agent in the CRS chooses a plan at random the first time it enters an intersection, uses a few simple productions to choose a plan in a previously visited intersection, and then uses Soar's planning mechanism to set plans in previously visited intersections after the object of interest has been located. Additionally, Kelley et al. discussed how they experienced difficulty in keeping the robot in good positions for autonomous intersection detection and with selecting thresholds to be used in rules that made decisions about the presence of intersections. Our Soar agent uses the operators described in Section 6.3.4.1 to maximize the likelihood the robot will be in a good position to detect intersections; however, it is possible that the CRS would have experienced trouble autonomously detecting intersections in the maze environment used for the SS-RICS test. The use of Hough transforms to estimate the hall width has allowed the fuzzy logic system in the CRS to detect intersections to adapt to different width hallways.

The use of a local metric representation (an occupancy grid) as a source of information to detect symbolic intersection information and generate the global topological structure of an environment has previously been described by Beeson et al. [92]. Intersections detected while following a previously prescribed path for exploring an environment were used to generate the global topological structure of the environment, which was then used to create an accurate global metric map. The algorithm developed to detect intersections did not label each intersection as a specific type (i.e. 'T' or 'R'), but was designed to be able to detect arbitrarily-shaped intersections. This system was also able to maintain a tree of possible topological maps and the ability of the system to generate the correct topological structure for an environment with nested loops was demonstrated. The ability to deal with loops and detect arbitrary intersections has not been implemented yet in the CRS; however, the CRS has been able to use knowledge about each type of detected intersection to make decisions about where to navigate in an environment and to create a symbolic representation of the environment with meaningful labels for the detected intersections. The methods described by Beeson et al. to generate global topological maps in the presence of loops could be useful in the future for Soar agents that are able to recognize topological loops in an environment.

Conclusion

7.1 Summary

The Cognitive Robotic System has been developed to use the Soar cognitive architecture for the control of mobile robots. The motivation for developing robot systems with cognitive capabilities and the characteristics of cognitive architectures that are useful for generating robot behavior with cognitive characteristics were discussed in Chapters 1 and 2. The specific mechanisms of the Soar cognitive architecture that make it an interesting architecture for robot control were also discussed. Chapter 3 presented the hardware, including the sensors, processors, and robotic platform, used for this research. The sensors used were inexpensive and provided limited information about the robot's environment when compared to most modern robotic platforms, but these sensors could be replaced or complemented with more advanced sensors such as stereo cameras and laser rangefinders. Software systems useful for motor control and sensor processing used to complement the weaknesses of Soar for unmanned vehicle control were described in Chapter 4. These sensor processing algorithms including computer vision, robotic mapping, and fuzzy logic systems and were useful for generating useful high-level symbolic information from the relatively minimal sensor information available in the CRS.

Implementations of the CRS have been developed for two different missions (as well as on two different robots). Chapter 5 described an implementation of the CRS for an outdoor navigation mission that had minimal sensor processing capabilities,

but was useful for gaining experience about how Soar could be useful for the control of robots. Despite having a minimal representation of its environment, the Soar agent was able to successfully navigate a complex cul-de-sac obstacle. During this mission, the agent demonstrated reflective behavior as it was able to change its strategy when its current method of avoiding obstacles could not be used to successfully navigate away from an obstacle. The results presented in Chapter 5 also showed the potential that increasing the knowledge in Soar (both about the environment and about how to act through the use of operators) has for generating more useful behavior.

The implementation of the CRS for the indoor search mission described in Chapter 6 incorporated several algorithms that would be either difficult or undesirable to implement in Soar, including common robotic mapping and computer vision algorithms. The Soar agent used for this mission used symbolic information generated by these algorithms to learn a topological map of the environment in Soar. The creation of this internal model helped the agent use Soar's planning and learning mechanisms to make intelligent decisions about how to navigate in this building environment. The agent was also able to generate and share directions describing a path to the location of an object of interest based on intersections in the environment.

7.2 Conclusions

The CRS has demonstrated how Soar can be integrated with common robotic motor and perceptual systems that complement the strengths of Soar for unmanned vehicles. The perceptual systems used occupancy grid, computer vision, and fuzzy logic algorithms to generate symbolic information about the environment that is useful for reasoning by the Soar agent. The ability to ground this symbolic information used by the Soar agent to objects in the robot's environment [96] was vital to the usefulness of Soar for generating intelligent behavior in this research. Also, while other researchers have also incorporated non-cognitive systems (including occupancy grids and computer vision algorithms) into systems that use cognitive architectures [23], [49], [51], no systems that have combined the use of a cognitive architecture with SIFT for object recognition, a fuzzy logic system, or a

topological map that represents connections between meaningful landmarks such as intersections are known.

Chapter 1 discussed how autonomy and intelligence are useful characteristics for unmanned vehicles. The CRS was able to demonstrate some of the aspects of these characteristics during the tests described in Chapters 5 and 6. The CRS performed these tests autonomously, i.e. “without any form of external control [2].” The CRS also demonstrated some of the aspects of intelligence mentioned in Chapter 1. Soar’s reasoning capabilities were used to perform a combination of reactive, deliberative, and reflective behaviors during the tests in Chapters 5 and 6. In the indoor search mission, the CRS was able to learn the topological structure of its environment. The Soar agent used this structure to generate an internal model of its environment that could be used for planning. The agent was able to learn from its experiences by saving the results of its planning as learned chunks that could be used to transition from deliberative to reactive behavior. The CRS was also able to communicate the information it learned about its environment through the generation of instructions to the object of interest using intersections as landmarks. Lastly, the Soar agent used a large set of knowledge (productions) to generate intelligent, context-specific behavior. However, there are aspects of intelligent behavior, such as the ability to think abstractly or comprehend complex ideas, that were not needed for the CRS to successfully complete the tests described in this dissertation.

The Cognitive Robotic System developed for this research and the Soar architecture are flexible enough to work on different unmanned vehicle missions. The CRS has been used for two missions, using different sensors, robots, Soar agents, and sensor processing and motor systems. The successful completion of the indoor search mission demonstrated that the CRS can use its navigation and map building capabilities to perform a higher-level task, such as intelligently searching an environment. Since navigation and map building capabilities are vital for most mobile robot missions, the indoor search mission has shown that the Soar agent can successfully use these capabilities to intelligently search a building, and Soar is capable of generating intelligent behavior, it is likely that, with the appropriate information from the environment and a Soar agent with the relevant knowledge, the CRS could act as a general purpose control system for mobile robots.

There are several aspects of using Soar that proved to be useful for developing the CRS. First, the Soar Markup Language makes it fairly simple to interface the Soar architecture with other software systems for sensor input and processing as well as motor control. By taking advantage of the processes defined by the Soar architecture, development efforts could be focused on developing productions that generate behavior, and not the software needed to perform functions such as matching relevant rules.

Soar's automatic subgoal mechanism that can be used for the hierarchical execution of complex operators is an architectural mechanism that is quite useful for the development of large agents. The low-level operators that an agent uses to perform a complex operator can be easily grouped together for use in a subgoal. This can help encourage the reuse of these rules (e.g. the low-level operators used to implement the navigate operator are the same in both the search and returnHome subgoals). Additionally, it allows the productions used to implement a complex operator to have a simpler set of conditions. For example, in the navigate subgoal, productions that propose the low-level operators to achieve the navigate operator can simply test that the agent is trying to implement the navigate complex operator, instead of including all of the conditions that resulted in the navigate operator being proposed.

Although using the sensors in the CRS to identify the intersections needed to generate the topological map in the Soar agent is challenging, once the agent has an internal model of its environment (and knowledge about the navigation plans that can be used to move through an intersection), Soar's planning mechanism was useful for making decisions about how to navigate through the CRS's environment. Additionally, the ability to create working memory elements with either i-support or o-support to take advantage of Soar's truth maintenance system was also useful for keeping the information stored in working memory current.

Soar has proven to be a useful choice for use in the CRS, given its focus on generating intelligent behavior, as opposed to matching the specifics of human behavior. While there are certainly robot applications where other cognitive architectures that focus on detailed cognitive modeling, such as ACT-R, would be useful, there are aspects of the missions studied here that would be more difficult to perform using ACT-R rather than Soar (e.g. planning and performing reasoning

that uses lots of information about the agent’s state).

While some of the benefits of using Soar have been described throughout this dissertation, there can be a steep learning curve to learning to program in Soar. For example, it can be difficult to learn to take advantage of the ability to write very complicated productions or to effectively use the two types of persistence (i-support and o-support) for working memory elements. However, there are support and tools, including helpful tutorial and manual materials and a supportive user community, available for assistance in learning to use Soar. There is also a debugger that can be used to show the details of the Soar agent’s execution. Using the debugger slows down the execution of an agent’s decision cycle, but is very useful for looking at the contents of working memory and the productions that have been fired, and therefore understanding how a Soar agent’s productions are being used.

7.3 Future work

There are some limitations of the research described in this dissertation and many improvements that could be made to the CRS. This research was more interested in understanding how Soar can be used to generate useful behavior and capabilities for the control of unmanned vehicles than achieving real-time behavior with the CRS. However, Soar has previously been used in systems with real-time constraints [33], and other research has studied the reactivity of the Soar architecture for simulated robotic systems [47]. The simple sensors used in the two implementations of the CRS described in this dissertation provided limited information about the system’s environment. This limited information contributed to the challenge of reliably detecting useful symbolic information about the environment and also limited the detail the agent was able to include in its internal model. The indoor search mission was performed in a structured environment in which intersections could be used as landmarks; with additional sensing capabilities, the CRS could attempt to perform a similar mission in a less structured environment.

The CRS was developed with the intention that it be extended in the future; this could include the development of more advanced robotic sensing and motor control systems and more advanced Soar agents. A more advanced robot system, especially one with improved sensing capabilities that could be used to obtain

more detailed information about environment, could result in a more interesting and capable Soar agent. It would also be useful to study the use of Soar in a robotic system with real-time constraints.

There are also many interesting future improvements possible for the Soar agent in the CRS. The largest Soar agent described in this dissertation uses only about ten percent of the rules used in the TacAir-Soar project, so a much larger Soar agent can be developed thanks to the scalability of the Soar architecture. A more capable agent for the indoor search mission that used all of the information about the environment available in the current CRS could consider the possibility of loops and incorrect nodes and or connections on the agent's topological map. Additionally, a more interesting Soar agent would also become possible if the CRS is able to generate more information about the environment with additional sensors or sensor processing algorithms.

Use of Soar's new architectural mechanisms that have potential for increasing the usefulness of Soar for unmanned vehicles [13] could also be explored. Semantic memory could be useful for storing facts about the world (such as that when the robot is facing north, performing a right turn will result in the robot facing east) [47] that are currently stored in working memory.

As the CRS is used for more complicated environments or for longer time periods, adding more and more information to working memory could negatively impact the reactivity of the Soar agent as the Rete algorithm is not scalable with the size of working memory.

While the Soar agent developed for the indoor search mission was able to use Soar's chunking mechanism, this required the agent to develop an accurate internal model of the environment, something that can be very challenging in real environments. New learning mechanisms such as reinforcement learning and learning using episodic memory may be especially useful for Soar agents controlling robots in real environments as these mechanism do not require knowledge of an internal model to be successful.

Incorporating some of Soar's new learning mechanisms could also provide more than one way for a Soar agent to improve its behavior.

The Cognitive Robotic System could also provide a platform to study collaboration between robots, collaboration between robots and humans including robot

learning through instruction from a human, or to explore the implementation of biological theories of navigation on mobile robots.

Appendix A

Example BrainStem TEA program

This appendix describes a TEA program that could be used to obtain measurements from a sonar sensor described in Section 3.3.1 and place these measurements in the BrainStem module's ScratchPad memory, where the Java program can access them for use by the Soar agent and the occupancy grid algorithm. The TEA program shown below uses three bytes in ScratchPad memory. The current measurement from the sonar sensor (the time it takes the return pulse to be sensed, measured in number of microprocessor ticks) is stored in two bytes. The addresses of the two bytes are given by `result1` and `result1 + 1`. Another byte (with address `flag1`) is used to coordinate access to the two bytes used to store the current measurement between the TEA program that writes the measurements to ScratchPad memory and the Java program that reads the measurements.

While the TEA program can write the result to both bytes in a single command (shown in lines 35 and 54 of the program below), the Java program used to read the result can only read a single byte at a time. The `flag1` byte is used to avoid the scenario in which the Java program has retrieved the high byte of the sonar result, but the TEA program writes a new sonar result to both bytes before the Java program is able to read the low byte of the initial result. While the current scheme for coordinating access between the TEA and Java programs does not guarantee the sonar measurement will not be updated by the TEA program while the Java program is retrieving the measurement, no problems have been detected as a result of its use.

After obtaining the first measurement from the sonar sensor, the TEA program

enters an infinite loop in which it obtains a measurement from the sonar sensor using a function that is included in software for BrainStem to obtain measurement from the sonar sensor (lines 32 and 43). This measurement is then stored in the `result1` and `result1 + 1` bytes. When the mission has been completed or the Java program needs to stop the TEA program for another reason, the Java program can send a command to the BrainStem module to stop the execution of the TEA program.

```

1  /* sonar.tea */
2
3  /* Reads one sonar sensors and places the result in the ScratchPad
4     bytes result1. ScratchPad byte flag1 is used for control of
5     read/write of bytes result1 */
6
7  /* routines for timer, digital pin, and ScratchPad functions */
8  #include <aCore.tea>
9  #include <aDig.tea>
10 #include <aPad.tea>
11
12 /* include the SRF05 driving routines */
13 #include "../aSystem/aSRF05.tea"
14
15 /* set which digital pin is used for the SRF05 sonar sensor */
16 #define SONAR1 1
17
18 /* Sonar results are in microprocessor ticks,
19     which are 1.6 us long */
20
21 void main(char flag1)
22 {
23     int distance = 0;
24     char result1 = flag1 + 1;
25
26     /* Initialize flag1 in the scratchpad to 0, indicating there

```

```
27         is not a valid result in the result1 scratchpad byte */
28         aPad_WriteChar(flag1, 0);
29
30         /* Call the SRF05 read sensor function for the initial
31            reading */
32         distance = aSRF05Mode2_ReadInt(SONAR1);
33
34         /* Write result */
35         aPad_WriteInt(result1, distance);
36
37         /* Set flag1 to 1 to indicate result has been written */
38         aPad_WriteChar(flag1, 1);
39
40         while(1){
41             /* Read sensor by calling the SRF05 read
42                sensor function */
43             distance = aSRF05Mode2_ReadInt(SONAR1);
44
45             /* Wait until Java program is not accessing
46                result1 */
47             while((aPad_ReadChar(flag1)==0));
48
49             /* First, reset flag1 (set to low) to indicate a
50                result is being written to result1 */
51             aPad_WriteChar(flag1, 0);
52
53             /* Write result */
54             aPad_WriteInt(result1, distance);
55
56             /* Set flag1 to 1 to indicate result has
57                been written */
58             aPad_WriteChar(flag1, 1);
59
```

```
60         } /*end of while(1) loop*/  
61  
62    } /* End of main */
```

Operator hierarchy for indoor search agent

This appendix gives the structure of the entire operator hierarchy of the Soar agent described in Section 6.3. The hierarchy shows both complex and primitive operators. Complex operators have operators listed underneath them whose names are indented, while primitive operators do not have operators with indented names listed directly underneath them. The complex operators with asterisks at the end of their names (set-plan-knownNode and getToClosestNode) uses Soar's planning mechanism to select the primitive operators used to implement this complex operator.

Some operators listed in multiple places within this hierarchy are proposed and applied using the same productions. For example, the removeIntersection, updateHoughInfo, updateMapIntersection, and updateMapObject operators are listed in many places in the hierarchy below, but the same production rules are used to propose and apply these operators throughout the hierarchy. Additionally, both the search and returnHome complex operators use the navigate complex operator. The same productions are used to perform the navigate complex operator in both the search and returnHome subgoals.

initialize-hallNavigator

initialize-intersectionDetectionAndHoughInfo

recordIntersection

- recordObject
- search
 - navigate
 - avoidObstacle
 - noteIntersection
 - removeIntersection
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - wait
 - followMiddle
 - forward-left
 - forward-right
 - go-straight
 - noteIntersection
 - removeIntersection
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - followWall
 - forward-left
 - forward-right
 - go-straight
 - initialize-FollowWall
 - noteIntersection
 - recordSonarFollowDistance
 - removeIntersection
 - removeOldSonarDistanceToFollow
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - go-straight
 - maintainOrientation

- forward-left
- forward-right
- go-straight
- noteIntersection
- removeIntersection
- updateHoughInfo
- updateMapIntersection
- updateMapObject
- performPlan
 - calculate-goal-state
 - calculateForGoStraight
 - calculateForStraightPlan
 - calculateForTurnToReturnHomePlan
 - calculateForUndoTurnPlan
 - calculateUsingFrontSonar
 - calculateWithNoObstacleInFront
 - calculateWithObstacleInFront
 - calculateUsingHallWidth
 - calculateWithJustInterruptedPlan
 - calculateWithNoObstacleInFront
 - calculateWithObstacleInFront
 - calculateForTurnAround
 - calculateForTurnLeft
 - calculateForTurnRight
 - calculateForTurnToReturnHome
 - change-current-goal
 - change-goal-status
 - change-plan-status
 - confirm-intersection
 - interruptPlan
 - noteIntersection
 - removeIntersection
 - updateHoughInfo

```

    updateMapIntersection
    updateMapObject
    wait
goToWaypoint
    moveToWaypoint
        avoidObstacleGoStraightPlan
            interruptPlan
            noteIntersection
            removeIntersection
            updateHoughInfo
            updateMapIntersection
            updateMapObject
            wait
        go-straight-to-waypointLoc
        interruptWaypointPlan
        noteIntersection
        removeIntersection
        turn-to-waypointLoc
            noteIntersection
            removeIntersection
            turn-left
            turn-right
            updateHoughInfo
            updateMapIntersection
            updateMapObject
            wait
        updateHoughInfo
        updateMapIntersection
        updateMapObject
        wait
    stopPlanSideObstacle
    turn-to-waypoint
        noteIntersection

```

- removeIntersection
- turn-left
- turn-right
- updateHoughInfo
- updateMapIntersection
- updateMapObject
- wait
- updateMap
 - calculateConnectionLength
 - calculate
 - estimateNodeLocation
 - estimateForDeadendPlan
 - estimateForStraightPlan
 - estimateForTurnPlan
 - calculateConnectionLengthToStart
 - calculateDistanceToStart
 - estimateNodeLocation
 - estimateForDeadendPlan
 - estimateForStraightPlan
 - estimateForTurnPlan
 - record-alternativeNodeType
 - recordConnection
 - recordConnectionToStart
 - recordIntersectionID
 - recordNodeType
 - setup-newNode
 - updateConnectionLength
 - calculate
 - update
- set-plan
 - markIntersectionAsVisited
 - set-plan-deadend
 - turnAround

```

    set-plan-intersection
        go-straight
        turnAround
        turnLeft
        turnRight
    updateActivePlan
        continueActivePlan
        interruptActivePlan
returnHome
    decideInitialAction
        makeDecisionWithActivePlan
            decideForDeadendPlans
            decideForNonDeadendPlans
        makeDecisionWithActivePlan
            continueNavigation
            delayDecision
            setTurnAroundPlan
    endTest
        generateDirectionsToObject
            finishDirectionsObjectSeenAfterNode
            finishDirectionsObjectSeenInNode
            getToClosestNode*
                go-straight
                turnAround
                turnLeft
                turnRight
            setInitialPosition
        placeObjectOnTopoMap
            placeObjectAfterNode
            placeObjectInNode
        readDirectionsToObject
            readDirectionWithNodeType
            readDirectionWithNoNodeType

```

- readLookDirection
- readLookDirectionWithNodeType
- stopAtStartLocation
- navigate
 - avoidObstacle
 - noteIntersection
 - removeIntersection
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - wait
 - followMiddle
 - forward-left
 - forward-right
 - go-straight
 - noteIntersection
 - removeIntersection
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - followWall
 - forward-left
 - forward-right
 - go-straight
 - initialize-FollowWall
 - noteIntersection
 - recordSonarFollowDistance
 - removeIntersection
 - removeOldSonarDistanceToFollow
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
 - go-straight

- maintainOrientation
 - forward-left
 - forward-right
 - go-straight
 - noteIntersection
 - removeIntersection
 - updateHoughInfo
 - updateMapIntersection
 - updateMapObject
- performPlan
 - calculate-goal-state
 - calculateForGoStraight
 - calculateForStraightPlan
 - calculateForTurnToReturnHomePlan
 - calculateForUndoTurnPlan
 - calculateUsingFrontSonar
 - calculateWithNoObstacleInFront
 - calculateWithObstacleInFront
 - calculateUsingHallWidth
 - calculateWithJustInterruptedPlan
 - calculateWithNoObstacleInFront
 - calculateWithObstacleInFront
 - calculateForTurnAround
 - calculateForTurnLeft
 - calculateForTurnRight
 - calculateForTurnToReturnHome
- change-current-goal
- change-goal-status
- change-plan-status
- confirm-intersection
 - interruptPlan
 - noteIntersection
 - removeIntersection

```
    updateHoughInfo
    updateMapIntersection
    updateMapObject
    wait
goToWaypoint
    moveToWaypoint
        avoidObstacleGoStraightPlan
            interruptPlan
            noteIntersection
            removeIntersection
            updateHoughInfo
            updateMapIntersection
            updateMapObject
            wait
        go-straight-to-waypointLoc
    interruptWaypointPlan
    noteIntersection
    removeIntersection
    turn-to-waypointLoc
        noteIntersection
        removeIntersection
        turn-left
        turn-right
        updateHoughInfo
        updateMapIntersection
        updateMapObject
        wait
    updateHoughInfo
    updateMapIntersection
    updateMapObject
    wait
stopPlanSideObstacle
turn-to-waypoint
```

```

        noteIntersection
        removeIntersection
        turn-left
        turn-right
        updateHoughInfo
        updateMapIntersection
        updateMapObject
        wait
updateMap
    calculateConnectionLength
        calculate
        copyCurrentNodeLocationToReturnToStart
        estimateNodeLocation
            estimateForDeadendPlan
            estimateForStraightPlan
            estimateForTurnPlan
    calculateConnectionLengthToReturnHomePlan
        calculateDistanceToReturnHomePlan
        estimateNodeLocation
            estimateForDeadendPlan
            estimateForStraightPlan
            estimateForTurnPlan
        estimateTurnAroundLocation
    record-alternativeNodeType
    recordConnection
    recordConnectionToReturnHomePlan
    recordIntersectionID
    recordNodeType
    setup-newNode
    startCalculatingDistFromNodeToStart
    updateConnectionLength
        calculate
        update

```

- setReturnPlan
 - markIntersectionAsVisited
 - set-plan-deadend
 - turnAround
 - set-plan-intersection
 - set-plan-knownNode*
 - go-straight
 - turnAround
 - turnLeft
 - turnRight
 - set-plan-unknownNode
 - go-straight
 - turnAround
 - turnLeft
 - turnRight
- updateActivePlan
 - continueActivePlan
 - interruptActivePlan

Explanation of selected Soar productions

Examples of Soar productions from the agent described in Section 6.3 are given in this appendix. Productions that perform the functions of state elaboration, operator proposal, operator comparison, and operator application are discussed. Additionally, the chunk learned during the test described in Section 6.4.2.1 is given. There are also many examples of productions available in the Soar tutorial that is included with the Soar software [60].

C.1 State elaboration

The `hallNavigator*elaborate*houghDistance` production is used to move the distance between Hough lines from the Soar agent's input-link to another part of the agent's working memory if the Hough lines identified from the current occupancy have enough contributors that make it likely they accurately represent the walls in the robot's environment.

Line number 1 contains the name of the Soar production. Every production starts with a test of a state. Line 2 tests that there is a state named `hallNavigator`. This state is generated by the operator that is always applied first to initialize the agent, so this condition will be true once the agent has been initialized, even if the agent is executing an operator that is located deep in the agent's operator hierarchy. The conditions in line 3 and 4 identify elements in

working memory that will be used in later conditions and create the variables `<hough-input>` and `<hough>` that can be used to refer to these elements later within the rule. The conditions in lines 5 and 6 test there are more than 50 contributors for both the Hough line to the left of the robot and the line to the right of the robot. Line 7 tests that there is a working memory element containing the distance between the Hough lines and binds the value of that distance in the variable `<dist>`. If all the conditions in lines 2–7 are true, the production fires and the action in line 9 creates a working memory element that uses the variable `<hough>` as an identifier, `^distanceBetweenHoughLines` as an attribute, and the value in the `<dist>` variable as a value. The working memory element created as a result of this action have i-support, meaning the element will exist only as long as the conditions in lines 2–7 remain true.

```

1  sp {hallNavigator*elaborate*houghDistance
2    (state <s> ^name hallNavigator
3      ^io.input-link.hough <hough-input>
4      ^sensorInfo.hough <hough>)
5    (<hough-input> ^left.contributors > 50
6      ^right.contributors > 50
7      ^distanceBetweenLines <dist>)
8    -->
9    (<hough> ^distanceBetweenHoughLines <dist>)
10 }
```

This production provides a check of the accuracy of the information from the Hough transform before the agent uses the results of the Hough transform for additional reasoning. Instead of testing the working memory element containing the distance between the lines on the input-link (line 7 of the production), which is always present in working memory, the agent can test working memory element that contains the distance between the Hough lines created by the action in line 9, which is present only if the Hough transform results are likely to identify the walls in an image.

C.2 Operator proposal

The production shown in this section is used when the agent is trying to follow the middle of the hall in the subgoal to implement the navigate complex operator. This production proposes a primitive operator that is capable of placing a motor command on the Soar agent's output-link.

The conditions in line 3 and 4 of this production check that there is a state with a name `followMiddle` and that this state has a superstate with a name of `navigate`. This means that unlike the production in Section C.1, which had a condition that tests that the agent has a state with the name of `hallNavigator` that will always be true once the agent has been initialized, the current production can only be fired when the agent's processing is currently at the appropriate place in its operator hierarchy. Line 5 is used to identify the working memory element `<cycleID>` and line 6 matches a working memory element that contains a parameter that holds the amount of time the CRS should run its motors if this operator is selected for application.

Other elaborations in the `followMiddle` subgoal are used to create the `<cycleID>` WME and add the attributes to that identifier that are tested in lines 8–10. The condition in line 8 checks that the robot's location has been elaborated to be in the middle of the current hall, line 9 checks that the robot's orientation is aligned with the hall's orientation, and line 10 checks that the `<cycleID>` has a `^time` attribute.

If all the conditions in lines 3–10 are matched by information in working memory, the production fires and proposes a go-straight operator with acceptable and indifferent preferences (the `+` and `=` signs, respectively). The proposed operator also has attributes (turn and throttle settings and length of time the motors should be on) that would be used to generate motor commands if this operator is selected.

The `<cycleID>` and `<clock>` objects are created by an elaboration that has a condition that tests the current clock value on the agent's input-link. This means that when the value of the clock on the agent's input-link changes, the production that created the `<cycleID>` and `<clock>` working memory elements retracts and these i-supported elements are removed from working memory.

As shown in Section 6.4.1, the information on the Soar's input-link is not

updated until a motor command is processed by the CRS. This means that once this operator (or another primitive operator that can place motor commands on Soar's output-link) is applied, a motor command will be placed on the agent's output-link. Once the CRS retrieves and processes the motor command, it will update the information, including the clock value, on the Soar's input-link.

The removal of the `<cycleID>` and `<clock>` elements means that the conditions in lines 5 and 8–10 are no longer true, causing the production to retract. If the conditions are determined to be true using the new information from the environment on the agent's input-link, the production can be fired to propose the go-straight operator again.

```

1  sp {hallNavigator*navigate*followMiddle*propose*
2      go-straight*in-middle*aligned
3      (state <s> ^name followMiddle
4          ^superstate.name navigate
5          ^cycle <cycleID>
6          ^top-state.parameters.motorRunTimes.forward
7              <motor-time>)
8      (<cycleID> ^location in-middle
9          ^alignedWithHall yes
10         ^time <clock>)
11  -->
12      (<s> ^operator <op> + =)
13      (<op> ^name go-straight
14          ^steer center
15          ^throttle slow
16          ^motorRunTime <motor-time>)
17  }
```

C.3 Operator preference

The production in this section is used to compare two operators that have been proposed with acceptable preferences. This production is used in the situation

described in Chapter 6 using Figures 6.19–6.24. The condition in line 3 checks that the agent is in the **returnHome** subgoal, trying to implement the **returnHome** complex operator, and line 4 is used to identify the area of working memory that contains some information that can be used to make decisions about how the agent should return home. Lines 5 and 6 identify operators **<op1>** and **<op2>** that have been proposed (i.e. have acceptable preferences). The condition in line 9 ensures that operator **<op1>** is **decideInitialAction** and the condition in line 10 checks that operator **<op2>** is **navigate**. Line 7 checks that the agent has previously delayed making the initial decision about how to return home (this happens if the agent realizes it has located all of the desired objects of interest while it is performing a navigation plan to move through an intersection other than a deadend). Line 8 checks that the agent has determined that it has moved far enough using the **navigate** complex operator that if the agent were to turn around, there is enough distance between the robot and the closest intersection for enough information to be integrated into the occupancy grid for the CRS’s sensor processing algorithms to accurately identify the walls of the hallway before the robot enters that intersection. If the conditions in lines 3–10 are true, the production fires and creates a preference to prefer the **decideInitialAction** operator to the **navigate** operator. The preference created as a result of this action has i-support.

This production is an example of how a Soar agent can use knowledge about its current situation to create preferences that allow Soar to choose between multiple proposed operators. The Soar agent used for the indoor search mission also has a similar production that checks that the condition in line 8 is not true, and prefers the **navigate** operator over the **decideInitialAction** operator.

```

1 sp {elaborations*returnHome*prefer*decideInitialAction
2     -delay-over-navigate
3     (state <s> ^name returnHome
4         ^top-state.returnHomeInfo <returnHomeInfo>
5         ^operator <op1> +
6         ^operator <op2> +)
7     (<returnHomeInfo> ^madeInitialDecision delay
8         ^farEnoughFromEnterNavigateLocation yes)
9     (<op1> ^name decideInitialAction)
```

```

10      (<op2> ^name navigate)
11  -->
12      (<s> ^operator <op2> < <op1>)
13  }

```

C.4 Operator application

This section will describe a production that applies an operator that has been proposed to generate a navigation plan to turn left through an intersection. The conditions in lines 2, 3, and 7 check that the agent has a state with a **set-plan-intersection** name that has an operator with a **turnLeft** name. The rest of the lines identify elements in working memory that will be useful for generating a navigation plan in the actions of this production.

The actions of this production generate a plan with three goals as described in Section 6.3.4.2. The first goal is to go straight to the center of the intersection, the second goal is to turn, and the third goal is to go straight to move out of the intersection. The working memory elements such as the type of intersection that caused this plan to be created and the orientation and width of the last hall will be used to calculate the waypoint for each goal when the goal becomes active. The working memory elements created as a result of this action have o-support and will persist in working memory until another o-supported production removes them or the elements become unlinked from the Soar agent's state.

```

1  sp {hallNavigator*set-plan-intersection*apply*turnLeft
2      (state <s> ^name set-plan-intersection
3          ^operator <op>
4          ^top-state <ts>
5          ^intersectionType <int-type>
6          ^intersectionIDnumber <int-ID-num>)
7      (<op> ^name <op-name> turnLeft)
8      (<ts> ^navigationPlans <plans>
9          ^clockInfo.clock <time>
10         ^sensorInfo.hough <houghInfo>)

```

```

11    (<houghInfo> ^correctedHallTheta <hall-theta>
12          ^hallWidth <hall-width>)
13    (<plans> ^currentPlanNumber <plan-num>)
14  -->
15    (<plans> ^planID <new-plan>)
16    (<new-plan> ^action <op-name>
17          ^currentGoalNumber 1
18          ^numberOfGoals 3
19          ^planNumber (+ <plan-num> 1)
20          ^startTime <time>
21          ^status active
22          ^triggeringIntersection <int-type>
23          ^triggeringIntersectionIDnumber <int-ID-num>
24          ^lastHallTheta <hall-theta>
25          ^lastHallWidth <hall-width>
26          ^goalID <1st-straight-goal>
27          ^goalID <turn-goal>
28          ^goalID <2nd-straight-goal>)
29    (<1st-straight-goal> ^action goStraight
30          ^goalNumber 1
31          ^status future
32          ^type waypoint)
33    (<turn-goal> ^action <op-name>
34          ^goalNumber 2
35          ^status future
36          ^type waypoint)
37    (<2nd-straight-goal> ^action goStraight
38          ^goalNumber 3
39          ^status future
40          ^type waypoint)
41  }

```

C.5 Chunk

This section describes a chunk that was learned by the Soar agent during the test described in Section 6.4.2.1. These conditions show this chunk will only be used when the agent has a state with a selection name (line 3) that has an `evaluate-operator` selected. The conditions in line 9 and 13 check that a turn-Left operator is being evaluated when the agent's current node is 1 and its current compass orientation is west. Other conditions check that the desired state is node 0, that there is a connection on the topological map from node 1 to node 0 to the south, and that the agent has the knowledge that when facing west, a turn left plan will result in a compass orientation of south. When this chunk fires, the turn left plan is evaluated to be successful and the Soar agent avoids having to internally simulate the effects of the plan.

```

1  sp {chunk-1*d5645*opnochange*1
2      :chunk
3      (state <s1> ^name selection ^operator <o1> ^evaluation <e1>
4          ^top-state <t1>)
5      (<o1> ^default-desired-copy yes ^name evaluate-operator
6          ^superproblem-space <s2> ^superoperator <s3> ^evaluation <e1>
7          ^superstate <s4>)
8      (<s2> ^name setPlan-knownNode)
9      (<s3> ^name turnLeft)
10     (<e1> ^desired <d1>)
11     (<d1> ^node 0)
12     (<s4> ^name setPlan-knownNode ^robotInfo <r1>)
13     (<r1> ^currentCompassOrientation west ^currentNode 1)
14     (<t1> ^mapInfo <m1> ^parameters <p1>)
15     (<m1> ^topologicalMap <t2>)
16     (<t2> ^node <n1>)
17     (<n1> ^nodeID 1 ^connections <c1>)
18     (<c1> ^connectionID <c2>)
19     (<c2> ^toNode 0 ^directionFromCurrentNode south)
20     (<p1> ^turnCompassResults <t3>)
```

```
21      (<t3> ^west <w1>)
22      (<w1> ^turnLeft south)
23      -->
24      (<e1> ^symbolic-value success +)
25  }
```

Trace of planning

This appendix shows a trace taken from the Soar debugger during the test described in Section 6.4.2.2 at the point in the test labeled as ‘10’ in Figure 6.28. At point ‘10,’ the robot is facing south and has just recorded the detection of the ‘Tr’ intersection corresponding to node 3 in Figure 6.29. The numbers at the left of some lines in the trace below (4862–4880) correspond to the number of the decision cycle in Soar in which an operator was applied or an impasse was generated. In the decision cycles preceding this trace, the ‘Tr’ intersection was recorded using the `recordIntersection` operator in the top state of the Soar agent. After this operator was applied, the `returnHome` operator was selected. Since it is a complex operator, an operator no-change impasse occurs, creating a subgoal with a state name of `returnHome`, in which the `setReturnPlan` operator is selected. In the subgoal created after the operator no-change impasse that occurred in decision cycle 4865, the agent realized that the intersection had already been visited (cycle 4866) and proposed an operator to set a navigation plan (cycle 4869). Since `set-plan-intersection` is a complex operator, a subgoal is generated in decision cycle 4868 to select an operator to generate a plan. Since the agent has recognized that it has visited this intersection before, the complex `setPlan-knownNode` operator is selected and another operator no-change impasse occurs.

In the subgoal generated to implement the `setPlan-knownNode` operator, an operator tie impasse occurs when the agent can not select between an operator proposed to set a go straight plan and an operator proposed to set a turn left plan. In the state (subgoal) generated to resolve the operator tie impasse, Soar’s planning

mechanism is used to resolve the impasse. Two **evaluate-operator** operators are proposed, one for each of the tied operators. The operator to evaluate the go straight plan is selected (cycle 4872), an operator no-change impasse occurs and a state in which to perform the evaluation is generated. A copy of an abstraction of the robot's state (current node of 3, current compass direction of south) is created and an internal simulation of the go straight operator is performed. As a result of the application of the simulated operator, the copy of the robot's state is changed to be at node 1, facing south. At this point, the agent is still not able to evaluate the go straight plan as being a success or a failure. To complete the evaluation, the agent then considers the relevant plans available when it is in node 1, facing south. Another operator tie impasse occurs (cycle 4875) between operators proposing go straight and turn left plans. The **evaluate-operator** corresponding to the turn left plan is selected. The turn left plan is internally simulated on a (second) copy of the agent state (node 1, facing south). The simulated operator leads to a dead end and an evaluation of failure for that plan. This evaluation is used to resolve the operator tie impasse generated in cycle 4875 and select the go straight plan for simulation. This simulation is performed on the first copy of the state and changes this copy of the state to node 0, facing south. Since node 0 is the desired state, the evaluation of the go straight operator is success. This evaluation is used to resolve the the operator tie impasse generated in cycle 4871; the agent is able to select the operator to set the go straight plan to move through node 3. After decision cycle 4880, the agent starts to perform the go straight plan.

Although this trace does not reflect it, the Soar agent learned a chunk for each of the three operator evaluations. It can be seen that the agent turns on the chunking mechanism only in states that are created to evaluate operators.

```

4862: 0: 05137 (returnHome)
4863: ==>S: S3765 (operator no-change)
4864:    0: 06904 (setReturnPlan)
4865:    ==>S: S3767 (operator no-change)
4866:        0: 06906 (markIntersectionAsVisited)
4867:        0: 06905 (set-plan-intersection)
4868:        ==>S: S3769 (operator no-change)
4869:            0: 06908 (setPlan-knownNode)

```

4870: ==>S: S3771 (operator no-change)
 4871: ==>S: S3773 (operator tie)
 4872: O: 06912 (evaluate-operator)
 4873: ==>S: S3775 (operator no-change)
 force-learn in state S3775.
 4874: O: C6124 (goStraight)
 Applied simulated goStraight, moved to node 1, facing south.
 4875: ==>S: S3777 (operator tie)
 4876: O: 06921 (evaluate-operator)
 4877: ==>S: S3779 (operator no-change)
 force-learn in state S3779.
 4878: O: C6129 (turnLeft)
 Applied simulated turnLeft, moved to node 2, facing east.
 Goal evaluate-operator failed.
 Goal setPlan-knownNode failed.
 Evaluation of 06919 (turnLeft) is failure
 4879: O: 06917 (goStraight)
 Applied simulated goStraight, moved to node 0, facing south.
 Goal evaluate-operator succeeded.
 Goal setPlan-knownNode succeeded.
 Evaluation of 06909 (goStraight) is success
 4880: O: 06909 (goStraight)

Bibliography

- [1] TRAFTON, J. G., A. C. SCHULTZ, N. L. CASSIMATIS, L. M. HIATT, D. PERZANOWSKI, D. P. BROCK, M. BUGAJSKA, and W. ADAMS (2006) “Communicating and Collaborating with Robotic Agents,” in *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation* (R. Sun, ed.), Cambridge University Press, Cambridge, MA, pp. 252–278.
- [2] BEKEY, G. (2005) *Autonomous Robots: From Biological Inspiration to Implementation and Control*, MIT Press, Cambridge, MA.
- [3] SULTANIK, E. A., I. BRAUDE, P. THAI, R. N. LASS, D. N. NGUYEN, J. B. KOPENA, W. C. REGLI, S. A. LISSE, S. N. FURTWANGLER, and A. J. VAYDA (2008) “Human-robot collaboration for remote surveillance,” in *Proceedings of the National Conference on Artificial Intelligence*, AAAI Press, Chicago, IL, pp. 1886–1887.
- [4] THRUN, S., M. MONTEMERLO, H. DAHLKAMP, D. STAVENS, A. ARON, J. DIEBEL, P. FONG, J. GALE, M. HALPENNY, G. HOFFMANN, K. LAU, C. OAKLEY, M. PALATUCCI, V. PRATT, P. STANG, S. STROHBAND, C. DUPONT, L. JENDROSSEK, C. KOELEN, C. MARKEY, C. RUMMEL, J. VAN NIEKERK, E. JENSEN, P. ALESSANDRINI, G. BRADSKI, B. DAVIES, S. ETTINGER, A. KAEHLER, A. NEFIAN, and P. MAHONEY (2006) “Stanley: The robot that won the DARPA Grand Challenge,” *Journal of Field Robotics*, **23**(9), pp. 661–692.
- [5] URMSON, C., J. ANHALT, D. BAGNELL, C. BAKER, R. BITTNER, M. N. CLARK, J. DOLAN, D. DUGGINS, T. GALATALI, C. GEYER, M. GITTELMAN, S. HARBAUGH, M. HEBERT, T. M. HOWARD, S. KOLSKI, A. KELLY, M. LIKHACHEV, M. MCNAUGHTON, N. MILLER, K. PETERSON, B. PILNICK, R. RAJKUMAR, P. RYBSKI, B. SALESKY, Y. SEO, S. SINGH, J. SNIDER, A. STENTZ, W. R. WHITTAKER, Z. WOLKOWICKI, J. ZIGLAR, H. BAE, T. BROWN, D. DEMITRISH, B. LITKOUHI, J. NICKOLAOU, V. SADEKAR, W. ZHANG, J. STRUBLE, M. TAYLOR, M. DARMS,

- and D. FERGUSON (2008) “Autonomous driving in urban environments: Boss and the Urban Challenge,” *Journal of Field Robotics*, **25**(8), pp. 425–466.
- [6] FLETCHER, L., S. TELLER, E. OLSON, D. MOORE, Y. KUWATA, J. HOW, J. LEONARD, I. MILLER, M. CAMPBELL, D. HUTTENLOCHER, A. NATHAN, and F. KLINE (2008) “The MIT-Cornell collision and why it happened,” *Journal of Field Robotics*, **25**(10), pp. 775–807.
- [7] JACKEL, L., E. KROTKOV, M. PERSCHBACHER, J. PIPPINE, and C. SULLIVAN (2006) “The DARPA LAGR program: Goals, challenges, methodology, and phase I results,” *Journal of Field Robotics*, **23**(11-12), pp. 945–973.
- [8] GOTTFREDSON, L. S. (1997) “Mainstream Science on Intelligence: An Editorial with 52 Signatories, History, and Bibliography,” *Intelligence*, **24**(1), pp. 13–23, also appeared in Wall Street Journal, Dec. 13, 1994.
- [9] ALBUS, J., R. BOSTELMAN, T. CHANG, T. HONG, W. SHACKLEFORD, and M. SHNEIER (2006) “Learning in a hierarchical control system: 4D/RCS in the DARPA LAGR program,” *Journal of Field Robotics*, **23**(11-12), pp. 975–1003.
- [10] KONOLIGE, K., M. AGRAWAL, M. R. BLAS, R. C. BOLLES, B. GERKEY, J. SOL, and A. SUNDARESAN (2009) “Mapping, navigation, and learning for off-road traversal,” *Journal of Field Robotics*, **26**(1), pp. 88–113.
- [11] KELLEY, T. and L. LONG (2010) “Deep Blue Cannot Play Checkers: The Need for Generalized Intelligence for Mobile Robots,” *Journal of Robotics*, **2010**, article ID 523757.
- [12] CHRISTENSEN, H. I., G. KRUIJFF, and J. WYATT (eds.) (2010) *Cognitive Systems*, COSMOS, Springer Verlag, Berlin, DE.
- [13] LAIRD, J. E. (2009) “Toward cognitive robotics,” in *Proceedings of SPIE*, Orlando, FL.
- [14] HUNTSBERGER, T. and A. STOICA (2010) “Envisioning cognitive robots for future space exploration,” in *Proceedings of SPIE*, Orlando, FL.
- [15] ALBUS, J. S. (2002) “4D/RCS A Reference Model Architecture for Intelligent Unmanned Ground Vehicles,” in *Proceedings of SPIE*, Orlando, FL.
- [16] HANFORD, S. D., O. JANRATHITIKARN, and L. N. LONG (2009) “Control of Mobile Robots Using the Soar Cognitive Architecture,” *Journal of Aerospace Computing, Information, and Communication*, **6**(2), pp. 69–91.

- [17] LONG, L. N., S. D. HANFORD, and O. JANRATHITIKARN (2010) "Cognitive Robotics using Vision and Mapping Systems with Soar," in *Proceedings of SPIE*, Orlando, FL, invited Paper.
- [18] LAIRD, J., A. NEWELL, and P. ROSENBLOOM (1987) "Soar: An Architecture for General Intelligence," *Artificial Intelligence*, **33**(3), pp. 1–64.
- [19] SUN, R. (2008) *The Cambridge Handbook of Computational Psychology*, 1 ed., Cambridge University Press, Cambridge, MA.
- [20] SMOLENSKY, P., L. NADEL, L. COOPER, P. CULICOVER, and M. HARNISH (1989) "Connectionist Modeling: Neural Computation/ Mental Connections," in *Neural Connections, Mental Computation*, MIT Press, Cambridge, MA.
- [21] NEWELL, A. (1990) *Unified Theories of Cognition*, Harvard University Press, Cambridge, MA.
- [22] LAIRD, J. (2008) "Extending the Soar Cognitive Architecture," in *Proceedings of the First Artificial General Intelligence Conference*.
- [23] KENNEDY, W., M. BUGAJSKA, M. MARGE, W. ADAMS, B. FRANSEN, D. PERZANOWSKI, A. SCHULTZ, and J. TRAFTON (2007) "Spatial Representation and Reasoning for Human-Robot Collaboration," in *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, AAAI, pp. 1554–1559.
- [24] KELLEY, T. D. (2006) "Developing a Psychologically Inspired Cognitive Architecture for Robotic Control: The Symbolic and Subsymbolic Robotic Intelligence Control System (SS-RICS)," *International Journal of Advanced Robotic Systems*, **3**(3), pp. 219–222.
- [25] ——— (2006) *Using a Cognitive Architecture to Solve Simultaneous Localization and Mapping (SLAM) Problems*, Technical Report ARL-MR-0639, Aberdeen Proving Ground, Aberdeen, MD.
- [26] KELLEY, T. D., E. AVERY, L. N. LONG, and E. DIMPERIO (2009) "A Hybrid Symbolic and Sub-Symbolic Intelligent System for Mobile Robots," in *AIAA InfoTech@Aerospace Conference*, AIAA, Wahington, DC, AIAA Paper No. 2009-1976.
- [27] BENJAMIN, P., D. LYONS, and D. LONSDALE (2004) "Designing a Robot Cognitive Architecture with Concurrency and Active Perception," in *Proceedings of the AAAI Fall Symposium on the Intersection of Cognitive Science and Robotics*, AAAI Press.

- [28] LEHMAN, J. F., J. LAIRD, and P. ROSENBLOOM “A Gentle Introduction To Soar, an Architecture for Human Cognition: 2006 Update,” Available online: <http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf>, last accessed July 26, 2011.
- [29] TRAFTON, J. G., N. L. CASSIMATIS, M. D. BUGAJSKA, D. P. BROCK, F. E. MINTZ, and A. C. SCHULTZ (2005) “Enabling effective human-robot interaction using perspective-taking in robots,” *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans.*, **35**(4), pp. 460–470.
- [30] ANDERSON, J., D. BOTHELL, M. BYRNE, S. DOUGLASS, C. LEBIERE, and Y. QIN (2004) “An integrated theory of the mind,” *Psychological Review*, **111**(4), pp. 1036–1060.
- [31] KIERAS, D. E. and D. E. MEYER (1997) “Overview of the EPIC architecture for cognition and performance with application to human-computer interaction,” *Human-Computer Interaction*, **12**(4), pp. 391–438.
- [32] NEWELL, A. and H. SIMON (1972) *Human Problem Solving*, Prentice Hall, Englewood Cliffs, NJ.
- [33] JONES, R., J. LAIRD, R. NIELSEN, K. COULTER, R. KENNY, and F. KOSS (1999) “Automated Intelligent Pilots for Combat Flight Simulation,” *AI Magazine*, pp. 27–41.
- [34] WRAY, R., J. LAIRD, A. NUXOLL, D. STOKES, and A. KERFOOT (2005) “Synthetic adversaries for urban combat training,” *AI Magazine*, **26**(3), pp. 82–92.
- [35] ANDERSON, J. and G. BOWER (1972) *Human Associative Memory*, Winston and Sons, Washington, DC.
- [36] TAATGEN, N. and J. ANDERSON (2008) “Constraints in Cognitive Architectures,” in *The Cambridge Handbook of Computational Psychology* (R. Sun, ed.), Cambridge University Press, New York, NY, pp. 170–185.
- [37] CASSIMATIS, N. L. (2002) *A Cognitive Architecture for Integrating Multiple Representation and Inference Schemes*, Ph.D., Massachusetts Institute of Technology, Cambridge, MA.
- [38] SUN, R., E. MERRIL, and T. PETERSON (2001) “From implicit skills to explicit knowledge: A bottom-up model of skill learning,” *Cognitive Science*, **25**(2), pp. 203–244.

- [39] CHONG, R. (2004) *Inheriting Constraint in Hybrid Cognitive Architectures: Applying the EASE Architecture to Performance and Learning in a Simplified Air-Traffic Control Task*, Tech. Rep. AFRL-HE-WP-TR-2005-0120, Wright-Patterson AFB, Wright-Patterson AFB, OH.
- [40] TAATGEN, N. and J. R. ANDERSON (2010) “The Past, Present, and Future of Cognitive Architectures,” *Topics in Cognitive Science*, **2**(4), pp. 693–704.
- [41] LONG, L. N. and S. D. HANFORD (2007) “Evaluating cognitive architectures for intelligent and autonomous unmanned vehicles,” in *2007 AAAI Workshop WS-07-04*, AAAI Press, Vancouver, BC, Canada, pp. 44–47.
- [42] PEW, R. and A. MAVOR (eds.) (1998) *Modeling human and organizational behavior: Applications to military simulation*, National Academy Press, Washington, DC.
- [43] RITTER, F., N. SHADBOLT, D. ELLIMAN, R. YOUNG, F. GOBET, and G. BAXTER (2003) *Techniques for modeling human performance in synthetic environments: A supplementary review*, Technical Report HSIAC-SOAR-2003-01, Wright-Patterson AFB: Human Systems Information Analysis Center, Wright-Patterson AFB, OH.
- [44] MORRISON, J. (2004) *A review of computer-based human behavior representations and their relation to military simulations*, Tech. Rep. IDA Paper P-3845, Institute for Defense Analyses, Alexandria, VA.
- [45] GLUCK, K. and R. PEW (eds.) (2005) *Modeling Human Behavior with Integrated Architectures: Comparison, Evaluation, and Validation*, Lawrence Erlbaum Associates, Mahwah, NJ.
- [46] JONES, R., C. LEBIERE, and J. CROSSMAN (2007) “Comparing modeling idioms in ACT-R and Soar,” in *Proceedings of the 8th International Conference on Cognitive Modeling*, Ann Arbor, MI.
- [47] LAIRD, J. E., N. DERBINSKY, and J. VOIGHT (2011) “Performance Evaluation of Declarative Memory Systems in Soar,” in *Proceedings of the Behavior Representation in Modeling and Simulation conference*, Sundance, UT.
- [48] MILLER, G. (1956) “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information,” *Psychological Review*, **63**, pp. 81–97.
- [49] TRAFTON, J. G., M. D. BUGAJSKA, B. R. FRANSEN, and R. M. RATWANI (2008) “Integrating vision and audition within a cognitive architecture to track conversations,” in *3rd ACM/IEEE International Conference on*

Human-Robot Interaction, HRI 2008, Association for Computing Machinery, pp. 201–208.

- [50] LAIRD, J. and P. ROSENBLOOM (1990) “Integrating execution, planning, and learning in Soar for external environments,” in *Proceedings of the National Conference of Artificial Intelligence*.
- [51] LAIRD, J., M. HUCKA, E. YAGER, and C. TUCK (1991) “Robo-Soar: An integration of external interaction, planning, and learning, using Soar,” *IEEE Robotics and Autonomous Systems*, **8**(1-2), pp. 113–129.
- [52] TAMBE, M., W. JOHNSON, R. M. JONES, F. KOSS, J. E. LAIRD, P. S. ROSENBLOOM, and K. SCHWAMB (1995) “Intelligent agents for interactive simulation environments,” *AI Magazine*, **16**(1), pp. 15–39.
- [53] SMITH, P. and S. WILLCOX (2005) “Systems research for practical autonomy in unmanned air vehicles,” in *AIAA InfoTech@Aerospace Conference*, AIAA, Washington, DC, pp. 1602–1613.
- [54] SCHULTZ, A., W. ADAMS, and B. YAMAUCHI (1999) “Integrating exploration, localization, navigation, and planning with a common representation,” *Autonomous Robots*, **6**, pp. 293–308.
- [55] TRAFTON, J. G., A. M. HARRISON, B. FRANSEN, and M. BUGAJSKA (2009) “An embodied model of infant gaze-following,” in *Proceedings of the 9th International Conference of Cognitive Modeling*, Manchester, UK.
- [56] BENJAMIN, D. P., D. LYONS, and D. LONSDALE (2006) “Embodying a cognitive model in a mobile robot,” in *Proceedings of SPIE*, Boston, MA.
- [57] BENJAMIN, D. P. and D. LYONS (2010) “A cognitive approach to classifying perceived behaviors,” in *Proceedings of SPIE*, Orlando, FL.
- [58] AVERY, E., T. KELLEY, and D. DAVANI (2006) “Using Cognitive Architectures to Improve Robot Control: Integrating Production Systems, Semantic Networks, and Sub-Symbolic Processing,” in *Proceedings of the Behavior Representation in Modeling and Simulation conference*, Baltimore, MD.
- [59] KELLEY, T. D. and E. AVERY (2010) “A cognitive robotics system: The symbolic and subsymbolic robotic intelligence control system (SS-RICS),” in *Proceedings of SPIE*, Orlando, FL.
- [60] <http://code.google.com/p/soar/wiki/Downloads> last accessed July 25, 2011.

- [61] DERBINSKY, N., J. LAIRD, and B. SMITH (2010) “Towards Efficiently Supporting Large Symbolic Declarative Memories,” in *Proceedings of the 10th International Conference on Cognitive Modeling*, Philadelphia, PA.
- [62] NUXOLL, A. M. and J. E. LAIRD (2007) “Extending cognitive architecture with episodic memory,” in *Proceedings of the National Conference on Artificial Intelligence*, AAAI Press, Vancouver, BC, Canada, pp. 1560–1565.
- [63] NASON, S. and J. LAIRD (2005) “Soar-RL: integrating reinforcement learning with Soar,” *Cognitive Systems Research*, **6**(1), pp. 51–9.
- [64] LATHROP, S. and J. LAIRD (2007) “Towards Incorporating Visual Imagery into a Cognitive Architecture,” in *Proceedings of the 8th International Conference on Cognitive Modeling*, Ann Arbor, MI.
- [65] FORGY, C. (1982) “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem,” *Artificial Intelligence*, **19**(1), pp. 17–37.
- [66] Soar Technology, “Soar: A Comparison with Rule-based Systems,” Available online: <http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/SoarRBSComparison.pdf>, last accessed July 26, 2011.
- [67] Soar Technology, “Soar: A Functional Approach to General Intelligence,” Available online: <http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/SoarFunctionalOverview.pdf>, last accessed July 26, 2011.
- [68] <http://code.google.com/p/soar/wiki/SMLQuickStartGuide> last accessed July 26, 2011.
- [69] HUFFMAN, S. and J. LAIRD (1995) “Flexibly instructable agents,” *Journal of Artificial Intelligence Research*, **3**, pp. 271–324.
- [70] GUPTA, A. and L. N. LONG (2009) “Hebbian Learning with Winner Take All for Spiking Neural Networks,” in *Proceedings of the 2009 IEEE International Joint Conference on Neural Networks (IJCNN)*, IEEE.
- [71] KUO, B. C. and F. GOLNARAGHI (2003) *Automatic Control Systems*, 8th ed., John Wiley and Sons, Inc., Hoboken, NJ.
- [72] SIMON, D. (2006) *Optimal State Estimation: Kalman, H_∞ , and Nonlinear Approaches*, John Wiley and Sons, Inc., Hoboken, NJ.
- [73] LOWE, D. (2004) “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, **60**(2), pp. 91–110.

- [74] CARSTEN, J., A. RANKIN, D. FERGUSON, and A. STENTZ (2009) “Global planning on the mars exploration rovers: Software integration and surface testing,” *Journal of Field Robotics*, **26**(4), pp. 337–357.
- [75] <http://www.robocup.org/> last accessed July 26, 2011.
- [76] BALAGUER, B., S. BALAKIRSKY, S. CARPIN, and A. VISSER (2009) “Evaluating maps produced by urban search and rescue robots: lessons learned from RoboCup,” *Autonomous Robots*, **27**(4), pp. 449–464, published online August 28, 2009.
- [77] CARBONE, C., A. FINZI, A. ORLANDINI, and F. PIRRI (2008) “Model-based control architecture for attentive robots in rescue scenarios,” *Autonomous Robots*, **24**(1), pp. 87–120.
- [78] GOSSOW, D., J. PELLEENZ, and D. PAULUS (2008) “Danger Sign Detection Using Color Histograms and SURF Matching,” in *Proceedings of the 2008 IEEE International Workshop on Safety, Security and Rescue Robotics*, IEEE.
- [79] <http://www.superdroidrobots.com/> last accessed June 21, 2011.
- [80] <http://www.acroname.com/> last accessed June 21, 2011.
- [81] <http://www.robot-electronics.co.uk/> last accessed June 21, 2011.
- [82] <http://www.garmin.com/> last accessed June 21, 2011.
- [83] <http://www.nubotics.com/> last accessed June 21, 2011.
- [84] <http://www.logitech.com/> last accessed June 21, 2011.
- [85] BORENSTEIN, J., H. EVERETT, and L. FENG (1996) “Where am I?” *Sensors and Methods for Mobile Robot Positioning*, Technical report, University of Michigan.
- [86] SHEARER, J. L., B. T. KULAKOWSKI, and J. F. GARDNER (1997) *Dynamic Modeling and Control of Engineering Systems*, 2nd ed., Prentice Hall.
- [87] THRUN, S. (2008) “Simultaneous Localization and Mapping,” in *Robot and Cognitive Approaches to Spatial Mapping* (M. E. Jefferies and W. K. Yeap, eds.), Springer, Berlin/Heidelberg, pp. 13–41.
- [88] MORAVEC, H. and A. ELFES (1985) “High-resolution maps from wide-angle sonar,” in *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, IEEE, pp. 116–121.

- [89] SMITH, R. and P. CHEESMAN (1986) "On the representation and estimation of spatial uncertainty," *International Journal of Robotics Research*, **5**(4), pp. 56–68.
- [90] MONTEMERLO, M., S. THRUN, D. KOLLER, and B. WEGBREIT (2002) "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem," in *Proceedings of the AAAI National Conference on Artificial Intelligence*, AAAI.
- [91] HAHNEL, D., D. FOX, W. BURGARD, and S. THRUN (2003) "A highly efficient FastSLAM algorithm for generating cyclic maps of large-scale environments from raw laser range measurements," in *IROS. Proceedings of the Conference on Intelligent Robots and Systems*, vol. 1, pp. 206–211.
- [92] BEESON, P., J. MODAYIL, and B. KUIPERS (2010) "Factoring the Mapping Problem: Mobile Robot Map-building in the Hybrid Spatial Semantic Hierarchy," *The International Journal of Robotics Research*, **29**(4), pp. 428–459.
- [93] THRUN, S. (1998) "Learning metric-topological maps for indoor mobile robot navigation," *Artificial Intelligence*, **99**(1), pp. 21–71.
- [94] CHOWN, E., S. KAPLAN, and D. KORTENKAMP (1995) "Prototypes, Location and Associative Networks (PLAN): Towards a unified theory of cognitive mapping," *Cognitive Science*, **19**, pp. 1–51.
- [95] KUIPERS, B., D. G. TECUCI, and B. J. STANKIEWICZ (2003) "The Skeleton In The Cognitive Map: A Computational and Empirical Exploration," *Environment and Behavior*, **35**, pp. 81–106.
- [96] HARNAD, S. (1990) "The symbol grounding problem," *Physica D: Nonlinear Phenomena*, **42**(1-3), pp. 335–346.
- [97] TOMATIS, N. (2008) "Hybrid, Metric-Topological Representation for Localization and Mapping," in *Robot and Cognitive Approaches to Spatial Mapping* (M. E. Jefferies and W. K. Yeap, eds.), Springer, Berlin/Heidelberg, pp. 43–63.
- [98] NOURBAKHSI, I. (1998) "Dervish: an office-navigating robot," in *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, MIT Press, Cambridge, MA, USA, pp. 73–90, ACM ID: 292117.
- [99] KOENIG, S. and R. G. SIMMONS (1998) "Xavier: a robot navigation architecture based on partially observable Markov decision process models," in *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, MIT Press, Cambridge, MA, USA, pp. 91–122, ACM ID: 292120.

- [100] WANG, Y., D. CHEN, and C. SHI (2009) "Autonomous Navigation Based on a Novel Topological Map," in *2009 Asia-Pacific Conference on Information Processing*, Shenzhen, China, pp. 254–257.
- [101] BUSCHKA, P. (2005) *An investigation of hybrid maps for mobile robots*, Ph.D., Orebro University.
- [102] YAMAUCHI, B. and R. BEER (1996) "Spatial learning for navigation in dynamic environments," *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics: A Publication of the IEEE Systems, Man, and Cybernetics Society*, **26**(3), pp. 496–505, PMID: 18263050.
- [103] YAMAUCHI, B., P. LANGLEY, A. C. SCHULTZ, J. GREFFENSTETTE, and W. ADAMS (1998) *Magellan: An Integrated Adaptive Architecture for Mobile Robotics*, Technical Report Technical Report 98-2, Stanford University, Palo Alto, CA.
- [104] HERTZBERG, J. and A. SAFFIOTTI (2008) "Using semantic knowledge in robotics," *Robotics and Autonomous Systems*, **56**(11), pp. 875–877.
- [105] GALINDO, C., A. SAFFIOTTI, S. CORADESCHI, P. BUSCHKA, J. FERNANDEZ-MADRIGAL, and J. GONZALEZ (2005) "Multi-hierarchical semantic maps for mobile robotics," in *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE.
- [106] MARTÍNEZ MOZOS, O. and W. BURGARD (2006) "Supervised Learning of Topological Maps using Semantic Information Extracted from Range Data," in *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE.
- [107] ZENDER, H., O. MARTÍNEZ MOZOS, P. JENSFELT, G. KRUIJFF, and W. BURGARD (2008) "Conceptual spatial representations for indoor mobile robots," *Robotics and Autonomous Systems*, **56**(6), pp. 493–502.
- [108] THRUN, S., W. BUGARD, and D. FOX (2006) *Probabilistic Robotics*, MIT Press.
- [109] DUDA, R. O. and P. E. HART (1972) "Use of the Hough Transformation to Detect Lines and Curves in Pictures," *Comm. ACM*, **15**(1), pp. 11–15.
- [110] TRUCCO and A. VERRI (1998) *Introductory Techniques for 3-D Computer Vision*, Prentice Hall.
- [111] YAMAUCHI, B. (2005) "The Wayfarer modular navigation payload for intelligent robot infrastructure," in *Unmanned Ground Vehicle Technology VII, March 29, 2005 - March 31, 2005*, vol. 5804 of *Proceedings of SPIE - The*

International Society for Optical Engineering, SPIE, Orlando, FL, United states, pp. 85–96, compendex.

- [112] BUSCHKA, P. and A. SAFFIOTTI (2002) “A virtual sensor for room detection,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2002*, vol. 1, IEEE, pp. 637–642.
- [113] LONG, L. N. (2008) “Scalable Biologically Inspired Neural Networks with Spike Time Based Learning,” in *ECSIS Symposium on Learning and Adaptive Behaviors for Robotic Systems, 2008. LAB-RS '08*, IEEE, pp. 29–34.
- [114] GUPTA, A. and L. N. LONG (2009) “Hebbian learning with winner take all for spiking neural networks,” in *Proceedings of the 2009 international joint conference on Neural Networks, IJCNN'09*, IEEE Press, Atlanta, Georgia, USA, pp. 1189–1195, ACM ID: 1704349.
- [115] ZADEH, L. (1965) “Fuzzy sets,” *Information and Control*, **8**(3), pp. 338–353.
- [116] LUGER, G. F. (2004) *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 5 ed., Addison Wesley.
- [117] ROSS, T. J. (2010) *Fuzzy Logic with Engineering Applications, Third Edition*, 3 ed., Wiley.
- [118] MIKOLAJCZYK, K. and C. SCHMID (2005) “A Performance Evaluation of Local Descriptors,” *IEEE Trans. Pattern Anal. Mach. Intell.*, **27**(10), pp. 1615–1630, ACM ID: 1083989.
- [119] BAY, H., A. ESS, T. TUYTELAARS, and L. VAN GOOL (2008) “Speeded-Up Robust Features (SURF),” *Comput. Vis. Image Underst.*, **110**(3), pp. 346–359, ACM ID: 1370556.
- [120] BAUER, J., N. SUNDERHAUF, and P. PROTZEL (2007) “Comparing several implementations of two recently published feature detectors,” in *Proceedings of the International Conference on Intelligent and Autonomous Systems*.
- [121] SE, S., D. LOWE, and J. LITTLE (2002) “Mobile Robot Localization and Mapping with Uncertainty using Scale-Invariant Visual Landmarks,” *The International Journal of Robotics Research*, **21**(8), pp. 735–758.
- [122] SE, S., D. G. LOWE, and J. J. LITTLE (2005) “Vision-based global localization and mapping for mobile robots,” *IEEE Transactions on Robotics*, **21**(3), pp. 364–375.
- [123] FOLKESSON, J. and H. CHRISTENSEN (2008) “SIFT Based Graphical SLAM on a Packbot,” in *Field and Service Robotics* (C. Laugier and R. Siegwart, eds.), vol. 42, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 317–328.

- [124] MEGER, D., P. FORSSEN, K. LAI, S. HELMER, S. MCCANN, T. SOUTHEY, M. BAUMANN, J. J. LITTLE, and D. G. LOWE (2008) “Curious George: An attentive semantic robot,” *Robotics and Autonomous Systems*, **56**(6), pp. 503–511.
- [125] <http://www.cs.ubc.ca/~lowe/keypoints/> last accessed June 21, 2011.
- [126] JANRATHITIKARN, O. (2007) *The Use of a Cognitive Architecture to Control a Six-Legged Robot*, M.S., The Pennsylvania State University, University Park, PA.

Vita

Scott D. Hanford

Scott Hanford was born on April 7, 1980 in Rochester, New York. He grew up in Macedon, New York with his parents, Pat and Keith Hanford, and his sister, Kim Hanford. Scott attended the University of Rochester in Rochester, New York where he graduated with a B.S. in Biomedical Engineering in May 2002. In 2004, Scott began graduate school at Penn State in Aerospace Engineering. That year he also married Amanda Danforth. His M.S. thesis was completed in 2005 and described the development of a stability augmentation system for a small unmanned quadrotor. His Ph.D. dissertation investigated using the Soar cognitive architecture for mobile robots and was completed in 2011. Scott and Amanda's first son, Noah, was born in November 2006. Their second son, Carter, was born in February 2011.