DAMN:  
A Distributed Architecture for  
Mobile Navigation  

Julio Kenneth Rosenblatt  

Submitted in partial fulfillment  
of the requirements of the  
Robotics Institute Ph.D. Program  

Thesis Committee:  
Charles Thorpe, Chair  
Martial Hebert  
Anthony Stentz  
David Payton (Hughes Research Labs)  

© 1997 by Julio Rosenblatt  
All rights reserved.
Acknowledgments

This research was funded primarily by DARPA under contracts “Technology Enhancements for Unmanned Ground Vehicles,” number DAAE07-96-C-X075, and “Unmanned Ground Vehicle System,” number DAAE07-90-C-R059, monitored by TACOM; “Perception for Outdoor Navigation,” number DACA76-89-C-0014, monitored by the US Army Topographic Center; and by the National Science Foundation under NSF grant BCS-9120655. The author was supported by a Hughes Research Fellowship.

There are many people I’d like to thank, or blame, for helping me to reach this point. Dave Payton started me off along this path, mentoring me, believing in me and treating me as a peer when I was still wet behind the ears. Together Dave and I formed a partnership that I will always remember with fondness, as I will all the wonderful people at the Hughes Artificial Intelligence Center and the camaraderie we shared there. The baton was passed to Chuck Thorpe, who had the patience to listen to my ideas and the tolerance to let me pursue them, giving me enough rope to do what I would. Not only did I learn much from Martial Hebert, he was also my favorite DAMN Guinea pig, if you’ll pardon my French. I am also indebted to Tony Stentz for always asking the right questions, planting new seeds of thought in my mind.

The Robotics Institute at Carnegie Mellon University cannot be beaten for its unbelievable resources, and it may sound trite, but the greatest wealth is of course the people there. Jim “Mr. Fixit” Frazer was charged with the impossible mission of keeping the Navlab vehicles running, and always did so with good spirits. Jim Moody, Kate Fissell, and Bill Ross, who collectively formed “IUS-Bugs,” provided incredible software support both in the lab and on the vehicles. Dirk Langer kept the controller alive, Barry Brumitt wrestled with the positioning system, and John Hancock kept the range finder limping on its last legs long enough for me to get some last results under my belt. Terry Fong helped with all of the above and also waded through a draft of this dissertation, for which I am especially grateful. The Navlab project is indeed a major team effort, and there countless others who also made significant contributions to it.

I’d also like to thank Dave Simon and Andrew Edie Johnson for putting me up with me as their officemate and for being good friends, and all my friends in Pittsburgh for making it a fun place to be, and especially Angeline Bon and Stephane Laveau for introducing me to my wife. Most of all, I’d like to thank Aleksandra for being the most caring and loving person on this planet, and without whom none of this would matter.
**TABLE OF CONTENTS**

Chapter 1: Introduction ............................................. 1
1.1 Mobile Robot Domain ........................................... 2
1.2 Architectures for Mobile Robot Control ...................... 4
1.3 Action Selection .................................................. 7
1.4 The Distributed Architecture for Mobile Navigation ....... 7
   1.4.1 Command Arbitration in DAMN .......................... 9
1.5 Summary .......................................................... 11

Chapter 2: Mobile Robot Control Architectures ................. 13
2.1 Architectural Issues ............................................. 13
   2.1.1 Deliberative vs. Reactive Reasoning .................... 14
   2.1.2 Centralized vs. Distributed Processing ................. 18
   2.1.3 Sensor Fusion vs. Command Arbitration ............... 21
   2.1.4 Top-down vs. Bottom-up Control ...................... 23
2.2 Architectural Solutions ......................................... 24
   2.2.1 Deliberative vs. Reactive Reasoning .................... 26
   2.2.2 Centralized vs. Distributed Processing ................. 27
   2.2.3 Sensor Fusion vs. Command Arbitration ............... 29
   2.2.4 Top-down vs. Bottom-up Control ...................... 30
2.3 Summary .......................................................... 31

Chapter 3: Action Selection ........................................... 33
3.1 Priority-Based Command Arbitration ......................... 33
   3.1.1 Limitations of Priority-Based Command Arbitration ... 35
3.2 Command Fusion ................................................ 38
   3.2.1 Orientation Selection Methods ......................... 38
   3.2.2 Vector Field Methods ................................... 42
### 3.3 DAMN Command Arbitration Methods

- **3.3.1 Constraint Arbitration**
- **3.3.2 Actuation Arbitration**
- **3.3.3 Effect Arbitration**

### 3.4 Limitations of Command Arbitration

- **3.4.1 Vote Semantics**
- **3.4.2 Control Issues**
- **3.4.3 Synchronization**
- **3.4.4 Representation of Uncertainty**

### 3.5 Utility Fusion

- **3.5.1 Vote Semantics**
- **3.5.2 Control Issues**
- **3.5.3 Synchronization**
- **3.5.4 Representation of Uncertainty**
- **3.5.5 Limitations of Utility Fusion**

### 3.6 Conclusion

### Chapter 4: DAMN Implementation

- **4.1 Arbiters**
  - **4.1.1 Turn Arbiter**
  - **4.1.2 Speed Arbiter**
  - **4.1.3 Field of Regard Arbiter**
  - **4.1.4 Path Arbiter**
  - **4.1.5 Coordination of Arbiters**
### Chapter 4: Development Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Behaviors</td>
<td>84</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Evolutionary System Development</td>
<td>85</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Safety Behaviors</td>
<td>86</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Movement Behaviors</td>
<td>89</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Goal-Directed Behaviors</td>
<td>91</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Combining Behaviors</td>
<td>95</td>
</tr>
<tr>
<td>4.3</td>
<td>Development Tools</td>
<td>95</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Creating Behaviors</td>
<td>96</td>
</tr>
<tr>
<td>4.3.2</td>
<td>User Interface</td>
<td>98</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Data Recording and Playback</td>
<td>99</td>
</tr>
<tr>
<td>4.4</td>
<td>Conclusion</td>
<td>100</td>
</tr>
</tbody>
</table>

### Chapter 5: Experimental Results

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Performance Metrics</td>
<td>103</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Mean Obstacle Proximity</td>
<td>103</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Mean Goal Distance</td>
<td>104</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Smoothness</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Vehicle and System Characteristics</td>
<td>105</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Vehicle Dynamics</td>
<td>105</td>
</tr>
<tr>
<td>5.3</td>
<td>Turn Arbiter: Vehicle Run Results</td>
<td>106</td>
</tr>
<tr>
<td>5.4</td>
<td>Path Arbiter: Vehicle Run Results</td>
<td>112</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Path Arbiter without Predictive Control</td>
<td>113</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Path Arbiter with Predictive Control</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Controlled Simulation Experiments</td>
<td>117</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Experimental Design</td>
<td>117</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Experimental Results</td>
<td>118</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Summary of Experimental Results</td>
<td>126</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusion</td>
<td>132</td>
</tr>
</tbody>
</table>
Chapter 6: Conclusion ................................................................. 133
  6.1 Architectures for Mobile Robot Control ......................... 133
  6.2 Contributions ................................................................. 135
    6.2.1 Novel Paradigm for Command and Utility Fusion .... 135
    6.2.2 Integration of Heterogenous Modules ..................... 140
  6.3 Future Work ................................................................. 141
    6.3.1 Definition of Uncertainties ................................. 141
    6.3.2 Definition of Utilities ........................................ 141
    6.3.3 Time-Dependent Planning ................................. 142
    6.3.4 Coordination of Multiple Degrees of Freedom .... 142
  6.4 Conclusion ................................................................. 143

Appendix A: Motor Schemas ................................................. 145
  A.1 Observed Prey-catching Behavior of Frogs ................. 145
    A.1.1 Schema Models ............................................. 147
    A.1.2 Comparison of Behavior Models ...................... 151

References ............................................................................. 155
LIST OF FIGURES

Chapter 1: Introduction

Figure 1-1 :Outdoor mobile robots. ............................................................................................5
Figure 1-2 :DAMN Arbiter and Behaviors. ....................................................................................9
Figure 1-3 :DAMN arbitration schemes. ......................................................................................10

Chapter 2: Mobile Robot Control Architectures

Figure 2-1 :Processing world input and state information to generate commands. ......................13
Figure 2-2 :Levels of planner deliberation. ..................................................................................15
Figure 2-3 :Recursive abstraction hierarchy. ................................................................................15
Figure 2-4 :Hybrid control architecture. .......................................................................................18
Figure 2-5 :Traditional functional decomposition of centralized systems. .................................19
Figure 2-6 :Hierarchical architecture with recursive functional decomposition. ........................19
Figure 2-7 :Hierarchical architecture with recursive task decomposition. ...................................20
Figure 2-8 :Levels of competence in a behavior-based architecture. ...........................................21
Figure 2-9 :Sensor fusion creates a centralized world model for planning. ................................22
Figure 2-10 :Command arbitration combines behavior outputs to produce action. ....................22
Figure 2-11 :Behaviors in Hughes architecture selectively activated from pool. .........................23
Figure 2-12 :Combining deliberative and reactive reasoning into one command space. ..............27
Figure 2-13 :Centralized arbitration of votes from distributed behaviors in DAMN. ..................28
Figure 2-14 :Tradition “planning levels” function at the same level in DAMN. ............................30

Chapter 3: Action Selection

Figure 3-1 :Combining behaviors in the Subsumption Architecture. ...........................................34
Figure 3-2 :Combinatorial circuits for action selection in the GAPPs architecture. ....................34
Figure 3-3 :Communication and arbitration among behaviors via a central blackboard. ............35
Figure 3-4: Prioritized behavior-based arbitration. .................................................................37
Figure 3-5: Hughes ALV system. ..........................................................................................39
Figure 3-6: Vector Field Histogram. .....................................................................................40
Figure 3-7: Command fusion via fuzzy logic. ........................................................................41
Figure 3-8: Component and combined vector fields. ............................................................43
Figure 3-9: Vector addition from potential field motor schemas. ..........................................44
Figure 3-10: Arbitration of maximum speed constraints. .....................................................45
Figure 3-11: Turn behaviors vote for candidate vehicle curvature commands. ......................46
Figure 3-12: Orientation selection process in DAMN. .........................................................48
Figure 3-13: Voting for field of regard polygons, mapped into pan-tilt space for arbitration. ....49
Figure 3-14: Control problems arising from dynamic state space. .......................................51
Figure 3-15: Effect of system dynamics on vehicle trajectory. .............................................52
Figure 3-16: Arbiter communicates candidate actions to each behavior. ..............................53
Figure 3-17: Unsynchronized behavior voting. .....................................................................54
Figure 3-18: Sources of uncertainty. ....................................................................................54
Figure 3-19: Map-based path arbiter voting. ........................................................................56
Figure 3-20: Utilities measures and uncertainties based on detection of detected features. .......57
Figure 3-21: DAMN Arbiter evaluates candidate actions using utility map information. ........57
Figure 3-22: Map-based path arbiter voting. ........................................................................58

Chapter 4: DAMN Implementation

Figure 4-1: Curvature-based turn command space. ...............................................................65
Figure 4-2: Turn arbiter command fusion process. ...............................................................66
Figure 4-3: Commanded speed proportional to maximum turn vote. .................................68
Figure 4-4: Constraint based speed arbitration. .................................................................69
Figure 4-5: Changing pan/tilt angles with motion of vehicle. ..........................................................69
Figure 4-6: Field of regard polygons mapped into pan/tilt command space. .................................70
Figure 4-7: Field of regard voting and arbitration. ........................................................................71
Figure 4-8: Continuous and discrete utility map representations. ..................................................73
Figure 4-9: Vehicle measurements defined from vehicle guide point to safety margin. ..........76
Figure 4-10: Examples of utility votes received from behaviors. ....................................................77
Figure 4-11: Integration of utilities along candidate trajectories. ..................................................78
Figure 4-12: Compute utility value for point $n$ along arc $A_{ij}$. ..................................................79
Figure 4-13: Coordination of turn and speed commands via behavior inputs. .............................82
Figure 4-14: Coordination of turn and speed arbiters via exchanged state information. ..............82
Figure 4-15: Coordination of turn and speed arbiters via consolidation. .....................................83
Figure 4-16: Two-dimensional coordinated speed and turn arbitration. .....................................83
Figure 4-17: Multiple arbiter nested control loop. ........................................................................84
Figure 4-18: Levels of competence in DAMN. ............................................................................85
Figure 4-19: Arc evaluation in the AVOID OBSTACLES behavior ............................................86
Figure 4-20: Avert Obstacles field of regard voting. ....................................................................87
Figure 4-21: Vehicle dynamics variables. ....................................................................................88
Figure 4-22: Evaluating a turn radius within SCARF. .................................................................90
Figure 4-23: Resampling of ALVINN output layer. .....................................................................91
Figure 4-24: Follow Path behavior. .............................................................................................92
Figure 4-25: Following Gradient Field pointers. ..........................................................................93
Figure 4-26: Using D* to evaluate distance from goal for each arc. ..........................................93
Figure 4-27: Grid-based utility based on distance to goal. ..........................................................94
Figure 4-28: Behavior module class hierarchy. ..........................................................................96
Figure 4-29: Graphical User Interface for turn arbiter. ...............................................................98
Figure 4-30: Map-based utility arbiter display. ...........................................................................99
Chapter 5: Experimental Results

Figure 5-1 :Steering actuator response characteristics. ...............................................................105
Figure 5-2 :System latencies in steering response. .......................................................................106
Figure 5-3 :Turn arbiter with Obstacle Avoidance and Seek Goals behaviors. .........................107
Figure 5-4 :Turn arbiter experimental run. ..................................................................................108
Figure 5-5 :Distribution of votes over time in turn arbiter and behaviors. .................................109
Figure 5-6 :Vote distributions at five specified locations along vehicle path ..............................110
Figure 5-7 :Path arbiter with Obstacle Avoidance and Follow Path behaviors. .........................112
Figure 5-8 :Vehicle curvature along on-road portion of vehicle run, w/o prediction. ...............113
Figure 5-9 :Vehicle run using path arbiter w/o prediction. ..........................................................114
Figure 5-10 :Path arbiter vehicle runs with predictive control. ..................................................115
Figure 5-11 :Vehicle curvature for path arbiter with prediction. ...............................................116
Figure 5-12 :Simulation testbed scenario. ....................................................................................118
Figure 5-13 :Typical vehicle path without system latencies. ......................................................118
Figure 5-14 :Mean obstacle proximity as a function of speed. ....................................................119
Figure 5-15 :Curvature profiles for turn and path arbiters at varying speeds. ............................120
Figure 5-16 :Effect of vehicle speed on path smoothness for each arbiter type. .......................121
Figure 5-17 :Commanded vs. actual curvature with actuator limit. .........................................121
Figure 5-18 :Commanded vs. actual curvature with system latency of 1 second. .....................122
Figure 5-19 :Degradation of turn arbiter performance with 1 second system latency. .............122
Figure 5-20 :Degradation of turn arbiter with 1 second latency and actuator limit. .................123
Figure 5-21 :Path metrics as a function of speed. .................................................................124
Figure 5-22 :Turn arbiter: paths executed at high speeds with 2 second latency. .................124
Figure 5-23 :Path arbiter w/o prediction: vehicle paths at high speeds. .................................125
Figure 5-24 :Path arbiter with prediction: actual and predicted paths at high speeds. .............125
Figure 5-25: Turn arbiter: effect of system latency on mean obstacle proximity. .......................126
Figure 5-26: Turn arbiter: effect of system latency on path smoothness. .................................127
Figure 5-27: Path arbiter w/o prediction: effect of latency on obstacle proximity. ......................128
Figure 5-28: Path arbiter w/o prediction: effect of latency on actual path smoothness. ..............129
Figure 5-29: Path arbiter w/o prediction: effect of latency on command smoothness. .................130
Figure 5-30: Path arbiter with prediction: effect of latency on mean obstacle proximity. ..........131
Figure 5-31: Path arbiter with prediction: effect of latency on smoothness. ............................132

Chapter 6: Conclusion
Figure 6-1: Centralized arbitration of votes from distributed behaviors in DAMN. ..................136
Figure 6-2: DAMN arbitration schemes. .....................................................................................138
Figure 6-3: Two-dimensional coordinated speed and turn arbitration. ......................................142
Figure 6-4: Multiple arbiter nested control loop. ......................................................................143

Motor Schemas 145
Figure A-1: Response of frog to various stimuli. ..............................................................146
Figure A-2: Frequency with which frog went through or around a fence. ......................146
Figure A-3: Orientation Selection model. .................................................................148
Figure A-4: Vector fields. ..................................................................................149
Figure A-5: Combined vector field for toad, prey, and barrier posts. ..............................150
Figure A-6: Vector field convergence map with peaks at goal locations. .........................150
Figure A-7: Vector Field Histogram. ........................................................................152
Figure A-8: Combining potential field motor schemas. .......................................................153
CHAPTER 1: INTRODUCTION

Experience over the years with different architectures and planning systems for mobile robots has led me to develop the approach described in this work in which an arbiter receives votes for and against commands from distributed subsystem modules and decides upon the course of action which best satisfies the current goals and constraints of the system. The Distributed Architecture for Mobile Navigation has been used to combine various systems of differing capabilities on several mobile robots; in addition to its use on the CMU Navlab vehicles, DAMN is also being used at Lockheed Martin, the Hughes Research Labs, and the Georgia Institute of Technology. DAMN arbiters have been used to integrate navigation modules for the steering and speed control of single as well as multiple vehicles at these sites, and have also been used to select field of regard for the control of a pair of stereo cameras on a pan/tilt platform. My work in building and testing DAMN has created a flexible and useful robot architecture that contributed significantly to the successes of the CMU Navlab project and the ARPA/OSD Demo II program. Vehicles under the control of DAMN drive at highway speeds, navigate across stretches of off-road terrain some kilometers in length, cooperate with other robotic vehicles, and accept commands from human supervisors; the types of data used in these systems include maps, color cameras, stereo cameras, sonars, ladars, and motion and position sensors.

This work investigates the issues involved in the design and construction of architectures for the control of autonomous systems in complex domains where uncertainty is a major factor and where real-time responsiveness is a necessity. An architectural framework has been developed that is demonstrated to satisfy key constraints for the class of domains under consideration and to have many important properties which facilitate the development of operational systems. The thesis of this dissertation is:

Centralized arbitration of votes from distributed, independent, asynchronous decision-making processes provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment.

Furthermore, a framework for developing and integrating independent decision-making modules communicating with such arbiters facilitates their development and leads to evolutionary creation of robust systems of incrementally greater capabilities.

The Distributed Architecture for Mobile Navigation (DAMN) represents an implementation of these ideas applied to the domain of mobile robot control. By combining elements of various approaches, DAMN has been able to overcome some of the limitations of each, thereby creating systems capable of real-time navigation in complex unstructured environments. DAMN has been successfully used to integrate a wide variety of navigation subsystems, creating complete mobile robot systems that perform tasks such as road following, cross-country navigation, and teleoperation while avoiding obstacles and meeting mission objectives.
1.1 Mobile Robot Domain

Mobile robots provide a useful and challenging domain in which to develop the thesis put forward here. Effective control of mobile robots and their associated sensors demands the synthesis and satisfaction of several complex constraints and objectives in real-time, particularly in unstructured, unknown, or dynamic environments such as those typically encountered by outdoor mobile robots. An architecture must connect the perception, planning, and control modules and provide a structure with which the system may be developed, tested, debugged, integrated and understood.

In addition to serving as an interesting case study for the development of control architectures, mobile robots have many uses, especially for gaining access to areas that are unreachable by or dangerous to humans such as cleanup of hazardous waste sites, inspection of nuclear power stations, military reconnaissance missions, subsea exploration, and planetary exploration. Other uses include the automation of routine tasks such as mail delivery and the inspection and waterproofing of heat tiles underneath the space shuttles. Recently, a new emphasis has been the incorporation of robotic technology in passenger vehicles in order to improve both the safety and efficiency of highway driving.

One of the key characteristics of most mobile robot domains is that uncertainty plays a large role, and that is the class of domains being studied in this work. Prior knowledge of the environment may be incomplete or not exist at all, and the environment may be dynamic, so that reasoning must be non-monotonic in nature and occur rapidly enough to be able to respond to unexpected events. Posterior knowledge gained via sensing is incomplete, inaccurate, and uncertain, as is knowledge of the effects of actions decided upon and taken by the system. In order to function effectively in unstructured, unknown, and dynamic environments, planning systems cannot generate a plan a priori that can be expected to perform reasonably in the face of such uncertainty, nor can they anticipate all contingencies that may arise. Planning systems must be reactive in the sense that their decisions must take into account current information and state at all times, proceeding in a data-driven manner, rather than attempting to impose unrealizable plans in a top-down fashion. Navigation of a mobile robot implies the meaningful progress towards and achievement of a set of goals, subject to the constraints imposed by the mission objectives, by the environment in which the vehicle is operating, and by the limitations of the information gathering and locomotion mechanisms themselves. Thus, a mobile robot architecture must combine deliberative goal-oriented planning with reactive sensor-driven behavior in order to be successful in a wide variety of navigation tasks.

Another important aspect of mobile robot systems is the need to combine information from several different sources such as video cameras, laser range finders, sonars, and inertial navigation systems; they must also be capable of combining objectives for a system that is to perform diverse tasks such as following roads, driving off-road, following designated paths, avoiding obstacles, and reaching goal destinations, as well as allowing for teleoperation. Because of the disparate nature of the raw sensor data and internal representations used by such
subsystems, combining them into one coherent system which combines all their capabilities has historically been difficult.

In a rapidly developing field such as mobile robot navigation, it is not reasonable to assume that all system components will be constructed from scratch, adhering to some strict guidelines of how they may perform their function. To be useful, an architecture must be readily usable by different programmers at different sites using different approaches. Therefore, the architecture should be able to accommodate and integrate subsystems that have been developed independently, so it must impose minimal restrictions on the nature of the data, representations, and algorithms used by these subsystems designed to achieve their respective tasks. It is also highly desirable to be able to add new subsystems without disrupting established functionality, thus providing for evolutionary development. In addition, the various sensors used will in general operate at different rates, as will the procedures that process their data, so they must be allowed to operate asynchronously to maximize the throughput and thus the responsiveness of the system.

When dealing with a physical system such as a mobile robot, it is also important to consider aspects of control such as stability and the limitations and constraints of the physical plant, such as the non-holonomic constraints of a wheeled vehicle, finite actuator capabilities, and the delays inherent in any system that arise from latencies in data acquisition, data processing, intermodule communications, and actuator response; together with the continuous motion of the vehicle, this implies that by the time the command is being executed the vehicle is no longer in the current state but actually in a future state, i.e. different position, heading velocity, turn rate, etc. An asynchronous distributed system presents an additional challenge in that, in general, the size of these latencies will be different for each module due to varying processing needs and sensor frame rates. If the various latencies of the system are not accounted for, the vehicle control will be unstable.

When designing a software architecture for the control of complex real-time systems such as mobile robots, there are many issues that must be addressed. First and foremost, the architecture must provide the means by which the system may accomplish its objectives efficiently; it must be able to satisfy real-time constraints, promote fault tolerance, and provide for the safety of the vehicle and of its surroundings. A second crucial consideration in the design of a mobile robot architecture is the ease with which a system may be developed, tested, debugged, and understood. In addition, it should not be overly restrictive so that a wide variety of independently developed modules may be integrated within its structure. It must provide a framework for sensing and reasoning processes to be conducted in a timely fashion within the context of purposeful, goal-oriented behavior. While many general purpose architectures are computationally equivalent or nearly so, the ease with which various classes of systems may be instantiated within them varies greatly. A crucial yet often neglected consideration in the design of a mobile robot architecture is the ease with which a system may be developed, tested, debugged, and understood, as well as adapted to novel domains and applications.
1.2 Architectures for Mobile Robot Control

To function effectively, an architectural framework for sensing and reasoning processes must be imposed to provide a structure with which the system may be developed, tested, debugged, and understood. However, the architecture should serve as an aid, not a burden, in the integration of modules that have been developed independently, so it must not be overly restrictive. It must allow for purposeful goal-oriented behavior yet retain the ability to respond to critical situations in real-time. Some key issues to be considered in the design of a planning and control architecture are whether the architecture should be centralized or distributed, whether the reasoning should be reactive or deliberative, and whether control should be top-down or bottom-up. In addition, there is a fundamental choice to be made in the method by which information from multiple sources is combined, via sensor fusion or command arbitration.

Some of the earliest mobile robots, such as Shakey the Robot [49] and the Stanford Cart [47], performed sensor fusion to create a world model, i.e., they operated by gathering all available sensory data and creating a unified representation of its environment. An Artificial Intelligence system would then generate plans, i.e. a pre-determined series of actions designed to attain a particular goal, that are provably correct and optimal, thus providing coherent goal-directed behavior; however, such plans are only correct to the degree that the world model is accurate. The plan thus created would then be handed to an execution module responsible for performing each step of the plan; the robot would then stop to gather more information and the process would repeat. Robotic systems of this sort had the advantage of being able to produce an optimal plan in a rational, coherent fashion, but they were limited in their usefulness due to their lack of reactivity to an uncertain or dynamic environment; robotic systems which operated in this manner were only capable of moving very slowly (~0.004 km/hr) and only in simple static environments.

Mobile robot systems were subsequently developed which were still highly centralized, but because they did not guarantee the optimality of their plans, they could process information at faster rates and therefore were capable of driving a vehicle at faster speeds. At Carnegie Mellon University (CMU), the Navlab vehicles, shown in Figure 1-1a and Figure 1-1b, provided a testbed for more complex capabilities for outdoor mobile robots, operating at higher speeds and on rougher terrain than the indoor robots that preceded them. A system developed on these platforms was capable of following roads and stopping when an obstacle was detected in the vehicle’s path [72]; this system was adapted to the DARPA Autonomous Land Vehicle (ALV) at Lockheed Martin, shown in Figure 1-1c, to demonstrate applications of the technology [74]. While such centralized systems allowed the vehicles they controlled to move at significantly faster speeds than their forebears (~4.0 km/hr), they were not responsive enough for more complicated maneuvers such as obstacle avoidance, and because of their monolithic nature they were difficult to maintain and to augment with new capabilities.

In order to overcome the limitations inherent in a purely centralized architecture and address the need for greater responsiveness in a mobile robot system, hierarchical architectures allowing distributed processing to occur were developed. The highest level of an hierarchical architecture performs in much the same way as the earlier systems; however, rather than a single monolithic
Figure 1-1: Outdoor mobile robots.

a) CMU Navlab 1; b) CMU Navlab II; c) DARPA Autonomous Land Vehicle
structure mapping these high level plans directly into motor-level commands, the highest level passes down its plans to intermediate levels that translate them into lower level commands, which are in turn passed to the level below. Within the NASREM architecture, developed at the National Bureau of Standards (currently NIST), each and every level had the same *SENSE-PLAN-ACT* structure, but operated at different scales of time and space, thus providing varying tradeoffs between assimilation and responsiveness [2]. While such an approach provides aspects of both deliberative planning and reactive control, the top-down nature of hierarchical structures tends to overly restrict the lower levels, thereby limiting the responsiveness of the system to unexpected external conditions; each layer assumes that its commands will be executed as expected by the lower levels, and must constantly monitor for and deal with violations of these assumptions [54].

As a response to the complexities and inefficiencies of centralized and hierarchical systems in unstructured, unknown, or dynamic environments, a new generation of behavior-based architectures emerged which were designed in a bottom-up fashion to provide greater reactivity to the robot’s surroundings [13]. In contrast to more traditional planners that build a world model and plan an optimal path through it, a behavior-based architecture consists of specialized task-achieving modules, or *behaviors*, that operate independently and are responsible for only a very narrow portion of vehicle control. Each behavior only receives as input that information specifically needed for its task, thus avoiding the need for sensor fusion and consequently its inherent bottlenecks. Behavior-based systems generate reactions, i.e. real-time responses to the current environment which *satisfice* ([67]) basic constraints necessary for the survival of the system. Distributed systems are less susceptible to catastrophic failure, and, decomposed by subtasks, a behavior-based architecture also facilitates the evolutionary creation of robust systems. However, such systems sacrifice generality and the ability to reason about the system’s own intentions and goals, so that their usefulness is restricted to simple domains where meta-reasoning is not required. In addition to their lack of internal structure, these architectures seek to minimize the use of internal representations, arguing that “the world is its own best model” [14], thereby improving responsiveness but adding unnecessary constraints to the system design and reducing its flexibility.

In order to achieve a symbiosis of deliberative and reactive elements, a hybrid architecture was proposed which incorporated a behavior-based system as the lower level of an hierarchical architecture, thereby providing the reactivity of the former and the control structure of the latter [52]. This Hughes architecture was successfully used to implement a system which was the first autonomous cross-country map and sensor-based system for a mobile robot [18]. Because of its ability to synthesize centralized and distributed approaches, the hybrid paradigm have become a common approach to designing robotic architectures [29]; DAMN belongs to this class of architectures.

Chapter 2 constitutes a more complete discussion of the trade-offs involved in choosing between deliberative vs. reactive reasoning, centralized vs. distributed processing, sensor fusion vs. command arbitration, and top-down vs. bottom-up control, and of the choices made in DAMN.
1.3 Action Selection

In a distributed architecture, it is necessary to decide which behaviors should be controlling the vehicle at any given time and to select among the candidate actions proposed by the behaviors. By appropriately fusing behavior commands through arbitration, a robot control system can respond to its environment without suffering the problems inherent in sensor fusion. Instead of performing sensor fusion, the system must combine command inputs to determine an appropriate course of action; the two most common means of achieving this are priority-based arbitration and potential fields.

In architectures which employ priority-based arbitration such as the Subsumption Architecture [13], action selection is achieved by assigning priorities to each behavior; of all the behaviors issuing commands, the one with the highest priority is in control and the rest are ignored. This allows for quick responses to new situations and stimuli, although, by definition, prioritization only allows one module to affect control at any given time. While this is an effective scheme for choosing among incompatible commands, it does not provide an adequate means for dealing with multiple goals that can and should be satisfied simultaneously. A compromise between behaviors cannot be achieved in such an all-or-nothing scenario; whenever one behavior’s output is overridden by another, the information and knowledge represented by that behavior is completely lost to the system.

A different means of action selection is to combine commands from various behaviors. Architectures that perform command fusion combine the commands from individual behaviors so that decisions may be made based on multiple considerations while preserving the modularity and reactivity of distributed systems. Potential fields is a method of planning robot trajectories based on combining the vector fields induced by schemas such as repulsive fields from obstacles and attractive fields from goals [7] [8] [37]; the force vector for the current robot position is calculated for each field and the sum of these vectors is then used to control the instantaneous motion of the robot. While the potential fields framework offers a means of fusing commands from multiple behaviors, it suffers from the well known problem of local minima. Another, perhaps more serious problem, is that arbitration via vector addition can result in a command which is not satisfactory to any of the contributing behaviors. Like the behaviors in a priority-based system, the output from each schema is only a single command; this output provides insufficient information to enable an arbiter to perform fusion beyond a simple averaging of commands. The issues involved in action selection are explored more deeply in Chapter 3, as are other related issues such as the semantics of the decision-making process, representational issues including discretization and uncertainty, and issues in the control of a dynamic system.

1.4 The Distributed Architecture for Mobile Navigation

Deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other’s deficiencies. Reactive components provide the basic capabilities which enable the robot to
achieve low-level tasks without injury to itself or its environment, while deliberative components provide the ability to achieve higher-level goals and to avoid mistakes which could lead to inefficiencies or even mission failure. Experience from implemented systems suggests that some form of layered control is essential to the development and use of a complex, versatile robot control system. Slower abstract reasoning is needed at the higher levels and faster numerical computations at the lower levels, thus allowing varying trade-offs between responsiveness and optimality as appropriate at each level. However, the nature of these levels and of the relationships between them is still an area being actively researched within the robotics community.

The symbolic manipulations of high-level planners have no direct grounding in the robot’s immediate reality, and is not well suited for use in control regimes with continuous variables, while lower-level controller calculations provide only very limited means of specifying and achieving goals. DAMN functions at the level of geometrical reasoning, which is essential to the successful performance of an autonomous mobile robot system. As such, it not only provides a language for modules which reason at that level, but it also serves as a bridge between the high-level reasoning of planning systems and the low-level numerical computations of controllers, and does so without imposing the constraints and assumptions of an hierarchical architecture. This structure has been dubbed a *free flow hierarchy* [75]. Behaviors are free to use whatever level of reactivity and deliberation is deemed appropriate, and all communicate with the arbiter in the same fashion regardless of the nature of that decision.

Likewise, centralized and distributed architectures each have important advantages. Centralized architectures provide the ability to coordinate, in a coherent fashion, multiple goals and constraints within a complex environment, while decentralized architectures offer the advantages of reactivity, flexibility, and robustness. The Distributed Architecture for Mobile Navigation is an implementation of the thesis that centralized arbitration of votes from distributed, independent, asynchronous decision-making processes provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment, and that a framework for developing and integrating independent decision-making modules communicating with such arbiters facilitates their development and leads to evolutionary creation of robust systems of incrementally greater capabilities.

The organization of the DAMN architecture is shown in Figure 1-2; it consists of a group of distributed reasoning modules communicating with a centralized command arbiter, sending votes in favor of actions that satisfy its objectives and against those actions which do not. The arbiter is then responsible for combining the modules’ votes and generating actions which reflects their objectives and priorities. Within the framework of DAMN, behaviors must be defined to provide the task-specific knowledge for controlling the vehicle. Each behavior in the system is responsible for a particular aspect of vehicle control or for achieving some particular task, receiving as input only that sensory information pertinent to the task, and processing that input to generate as output votes for and against possible actions. Voting takes places in a commonly defined action or state space, thus providing a uniform interface to the arbiter that is independent of the task the behavior
is concerned with, the level of planning at which it operates, or the algorithms and paradigms which it uses for its decision-making process. DAMN is designed so that various behaviors can be easily added or removed from the system, depending on the current task at hand.

Each behavior operates asynchronously and in parallel with other behaviors, sending its outputs to the arbiter at whatever rate is appropriate for that particular function. Each behavior is assigned a weight reflecting its relative priority in controlling the vehicle; however, this weighting is not a strict prioritization, thus votes from all behaviors are used in determining what the next action should be. A mode manager may also be used to vary these weights during the course of a mission based on knowledge of which behaviors would be most relevant and reliable in a given situation. These behavior votes are then periodically combined by the arbiter and the resulting command is sent to the controller.

Several arbiters may be instantiated and run concurrently within DAMN, each responsible for the control of a different aspect of the overall system. There is usually a one-to-one mapping between an arbiter and a degree of freedom to be controlled, such as vehicle steering or speed, but an arbiter may control multiple degrees of freedom when they are interdependent and must be jointly controlled. Each arbiter operates separately from all other arbiters, although they may communicate their outputs to one another for loose coordination as with speed and steering control. Likewise, behaviors are usually concerned only with the control of one particular degree of freedom and sends its votes to the corresponding arbiter, but a behavior may also send votes to more than one arbiter if appropriate, and for convenience an arbiter may also send multiple independent sets of votes to a single arbiter.

1.4.1 Command Arbitration in DAMN

Within DAMN, independent behaviors operate in a distributed fashion to generate votes for actions based on domain-specific knowledge, while a central arbiter combines their results to generate reasonable behavior which obeys all constraints and simultaneously satisfies as many objectives as possible by choosing that action which maximizes the sum of the behavior votes. In
order to preserve the respective advantages of centralized and distributed architectures, sufficient information must be communicated from the behaviors to allow the arbiter to make intelligent decisions, but the arbiter must not be so complex as to become a bottleneck for the system. Various points along this trade-off spectrum have been explored within DAMN, using different types of arbiters and vote structures. There are currently four different arbitration schemes in effect, illustrated in Figure 1-3.

The simplest DAMN arbitration scheme is constraint arbitration, in which each behavior determines the highest value of the command being determined that satisfies the constraint which it seeks to enforce; the behaviors send that maximum value to the arbiter, which then takes the minimum of these values, thereby simultaneously satisfying all constraints. For example, speed behaviors determine their maximum speed vote based on constraints such as avoiding vehicle tip-over and avoiding obstacles, and the arbiter goes at the speed of the slowest vote, as shown in Figure 1-3a.

In the actuation arbitration scheme, each behavior votes for or against various alternatives in the actuator controller’s command space; for example, the turn arbiter receives votes for a fixed set of vehicle turn radii, as shown in Figure 1-3b. The arbiter then sums these votes and selects that action which has the highest total score, and send the command to the controller.
In the effect arbitration scheme, each behavior votes for or against various alternatives in an abstract command space, i.e., they vote for the desired effect of the mechanism being controlled rather than the direct control of the mechanism’s actuators. For example, a field of regard arbiter and its associated behaviors have been implemented and used for the control of a pair of stereo cameras on a pan/tilt platform, as shown in Figure 1-3c.

In the map-based utility arbitration scheme, behaviors do not vote for commands but instead express the utility of possible world states which may be represented within a map, and it is the responsibility of the arbiter to determine which states are actually attainable and how to go about achieving them. This type of arbiter is no longer performing command fusion, nor is it performing sensor fusion; it is combining utilities to perform utility fusion. This new approach strikes a balance between the extremes of action selection and sensor fusion and has been found to yield many benefits. For example, a map-based path arbiter has been implemented as a means of voting for and producing steering control. Behaviors communicating with the path arbiter vote on the desirability of various possible vehicle locations, and the arbiter maintains a local map of these votes, as indicated in Figure 1-3d. The path arbiter then evaluates the candidate trajectories, and selects that one for which the total utility is the greatest.

Chapter 4 contains a detailed description of each of these DAMN arbiters, as well as of the behaviors that send votes to these arbiters, and Chapter 5 provides a description of complete systems that have been developed with these modules and the results obtained, as well as experiments conducted to study the effects of varying domain characteristics such as dynamics and the respective advantages of the various types of arbitration in differing conditions. Chapter 6 describes the contributions of this work, as well as further investigations that may be conducted along this line of inquiry.

1.5 Summary

Because reactivity is essential for any real-time system, we must eschew the sensing and planning bottlenecks of centralized systems, but if we are to avoid sensor fusion, the system must combine command inputs to determine an appropriate course of action. However, priority-based arbitration only allows one module to affect control at any given time. Command fusion provides a mechanism for the concurrent satisfaction of multiple goals, and allows modules to be completely independent, thus allowing incremental, evolutionary system development.

The Distributed Architecture for Mobile Navigation is a planning and control architecture in which a collection of independently operating behaviors collectively determine a robot’s actions. A command arbiter combines the behavior outputs and selects that action which best satisfies the prioritized goals of the system. The distributed, asynchronous nature of the architecture allows multiple goals and constraints to be fulfilled simultaneously, thus providing goal-oriented behavior without sacrificing real-time responsiveness. Unlike other behavior-based architectures, DAMN is designed so that behaviors provide both deliberative and reflexive capabilities; the use
of distributed shared control allows multiple levels of planning to be used in decision-making without the need for an hierarchical structure.

Thus, DAMN performs centralized arbitration of votes from distributed, independent, asynchronous decision-making processes and in so doing provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment. DAMN also provides a framework for developing and integrating independent decision-making modules communicating with such arbiters, thus facilitating their development and leading to evolutionary creation of robust systems of incrementally greater capabilities.

DAMN has been used to combine various systems of differing capabilities on several mobile robots, and has also been used for active sensor control. Various subsystems developed at CMU and elsewhere have been integrated within this architecture, creating systems that perform road following, cross-country navigation, map-based route following, and teleoperation while avoiding obstacles and meeting mission objectives.
CHAPTER 2: MOBILE ROBOT CONTROL ARCHITECTURES

In order to function in unstructured, unknown, or dynamic environments, a mobile robot must be able to perceive its surroundings and generate actions that are appropriate for that environment and for the goals of the robotic system. To function effectively, an architectural framework for these sensing and reasoning processes must be imposed to provide a structure for combining information from several different sources. The role of the mobile robot control system is to map perceptual and state information into action so as to achieve the desired effects, as shown schematically in Figure 2-1; however, the architecture should serve as an aid, not a burden, in the integration of modules that have been developed independently, so it must not be overly restrictive. It must allow for purposeful goal-oriented behavior yet retain the ability to respond to potentially dangerous situations in real-time while maintaining enough speed to be useful.

Figure 2-1: Processing world input and state information to generate commands.

In this chapter we explore the inherent problems of the domain, the nature of the trade-offs that must be considered, and describe some of the solutions that have been proposed and implemented in existing architectures. The solutions proposed in this work are then described, and the argument is made that centralized arbitration of votes from distributed, independent, asynchronous decision-making processes provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment. The the Distributed Architecture for Mobile Navigation (DAMN) is then described in the context of these trade-offs, and is used illustrate the concepts and benefits of the proposed architectural solutions.

2.1 Architectural Issues

Some key issues to be considered in the design of a mobile robot control architecture are whether the architecture should be centralized or distributed, whether the reasoning should be reactive or deliberative, whether input combination should occur via sensor fusion or command arbitration, and whether control should be top-down or bottom-up. These issues, which are interrelated and must be considered together, should not be treated as dichotomies, but rather as continuous spectra. While some architectures take extreme positions on these issues and have been demonstrated in very narrowly defined niches, pragmatism dictates that there is a vast middle ground to be explored; reasonable trade-offs can and should be considered in the design of a
system order to achieve its desired capabilities. The question then becomes not “which one?” but rather “how much of each?” for a particular class of domains. Let us explore each of these issues in turn.

### 2.1.1 Deliberative vs. Reactive Reasoning

An important design consideration when building a robot control system is balancing the need for carefully planned, theoretically optimal actions to achieve goals against the need for quickly decided, satisficing reactions to a dynamic, uncertain environment. Deliberative reasoning provides the ability to achieve higher-level goals and to avoid mistakes which could lead to inefficiencies, while reactive responses enable the robot to avoid injury to itself or its environment. The absence of either type of reasoning could result in the robot being unable to satisfy its objectives.

Another important consideration is the types of representations used and supported by an architecture. The structures defined play a large role in determining how appropriate a particular architecture may be for different classes of domains and tasks, if not in absolute capability then at least in ease of use. Like programming languages, there is no single correct answer; the suitability of a particular abstraction depends on the architecture on which it is implemented, the level at which the task is best described, the real-time demands of the task, and the skills and preferences of the system developer. Many architectures compel the systems built within their framework to adopt one position or the other on both the level of planning and the level of abstraction of the representations used; ideally, an architecture should provide for the effective, efficient inclusion of reasoning systems at any point along the deliberative-reactive spectrum as appropriate for the domain without forcing the programmer to circumvent architectural constructs to achieve the proper tradeoffs.

**Deliberative Planners**

Deliberative planners function by searching through a state space to find a sequence of operators that result in the goal state \( G \) being reached from the start state \( S \), as illustrated by the “Optimal Plan to Goal” in Figure 2-2. The earliest work in robot control architectures attempted to reason by manipulating abstract symbols using only first order predicate logic [49]; the world state was represented as a set of truth values for the objects of interest and their relevant properties. Likewise, the goal state consisted of a set of desired truth values. Actions were described by their preconditions, i.e., what had to be true before the action could be taken, and by their effects, a list of assertions to be added or deleted from the world state as result of the action taken. A resolution theorem prover, STRIPS [49], would then “prove” the achievement of the desired goal state from the initial state via a series of planned actions which it generated and then handed off to a run-time execution module. This was an important first step in the automatic creation of plans for robotic action, but it was immediately apparent that it suffered from some severe drawbacks. The most obvious shortcoming of this system was its slowness, due to the combinatorial explosion of the search process. The robots would need to stop and deliberate for a very long period of time before
they could move another step, and even on today’s computers real-time needs would not be met. Furthermore, no sensing or planning took place during the execution of a step, and the effects of an action were assumed to have occurred without verification, thus rendering the robots unresponsive to their environment.

As a result of these limitations, abstraction hierarchies were introduced as a mechanism for dealing with the exponential nature of the search space, as well as the limitations of \textit{a priori} knowledge. As shown in Figure 2-3, a plan to achieve the top level goal is determined by searching in a high level abstract search space; intermediate states of that plan can then be used as subgoals at a lower level of abstraction. This process recurses until primitive actions are composed into a directly achievable plan. This refinement of high level plans into primitive actions may either occur off-line within the planner, or on-line during execution with a functional hierarchy constructed to mirror the task hierarchy, as described in Section 2.1.2.

\textit{Interleaving Planning and Execution} 

In situations where there is not enough information available to plan a sequence of actions that will lead to the goal, or where the uncertainty is too great, the planner may instead expand the search only so far as a lookahead horizon that may be defined by either informational or
processing limitations. This “Optimal partial Plan”, shown in Figure 2-2, finds the optimal path to a subgoal $G_L$ at the search horizon lookahead depth. Such a scheme limits the combinatorial explosion of the full search space and allows for an iterative solution towards the goal which may be recomputed in real-time, taking into account the latest available information as it becomes available. Thus the need to compute the entire plan off-line is obviated, and instead it becomes possible to “interleave planning and execution”; i.e., search up until the lookahead horizon, execute that plan, search again from the ensuing vehicle state, and iterate until the goal is reached. In addition, maintenance goals that have no definite end may be dealt with in this manner.

Reactive Architectures

Rather than constructing a model, planning a course of action within that model, and mapping that plan into concrete actions, reactive architectures are designed to respond directly to sensory stimuli, using compiled procedural knowledge to instantaneously map perceptions to actions. As indicated by the “Reactive Plan” in Figure 2-2, reactive execution performs no lookahead and explicit evaluation of possible future states. Problems with uncertainty in perception are avoided by sensing at a rate rapid enough so that false readings have a very limited impact; problems with uncertainty in action are avoided by acting at all moments on the currently perceived world. No assumptions are made about the persistence of previously observed states. Explicit world models and knowledge-based planning are avoided under the belief that “the world is its own best model” [14]. However, this assumes that an intelligent agent’s sensors and the algorithms which process them are essentially free of harmful noise and that they can not benefit from evidence combination between consecutive percepts. In addition, disallowing the use of internal representations requires that all environmental features of immediate interest be visible to the vehicle sensors at all times, thus adding unnecessary constraints and reducing the flexibility of the overall vehicle system. Without internal representations and plans, there is also no means to direct the robot towards a goal unless the cues are directly observable; to compensate, reactive systems must wander around a great deal, essentially performing a random walk, until they happen to reach a point where the means of achieving a goal becomes apparent. Further contributing to their inefficiency, they have no means of avoiding or detecting local minima along the way.

The capabilities of a system that prohibits any internal state whatsoever, as well as any intermodule communication, was investigated [16]. A system was carefully constructed for the specific task of collecting coke cans in an office environment; however, its generality and robustness to changes in its environment were severely limited. Few reactive architectures are actually taken to such an extreme and maintain some degree of internal state, each finding various trade-offs in the reactive/deliberative spectrum. Behaviors in the Subsumption Architecture [13] contain finite state machines to remember which mode it is operating in along with some minimal state information, and a network of behaviors was defines that was capable of learning a topological map [45]. Motor Schemas [6] map perception directly into action but also maintain local state as needed to perform their task.
**Situated Action**

The situated action approach has been proposed as another means of representing the world and deciding responses directed towards achievement of goals which avoids the limitations of reactive agents because internal representations may be arbitrarily complex, but these representations are constrained to have a direct deictic referent in the immediate world [1]. Because no abstract symbols are used, the problem of object identification is reduced to the much more easily achievable task of object recognition. Where deliberative architectures use compiled symbolic knowledge of how to *act* according to a given plan and reactive architectures use compiled procedural knowledge of how to *react* to a given stimulus, situated agents use compiled knowledge of how to *interact* with a given environment [60]. Situated agents have been somewhat successful when applied to limited tasks in simulated domains, but their usefulness in real world applications has yet to be demonstrated. Both reactive planners and situated agents also suffer from their limited ability to represent and reason about possible future states of the world; in addition to possible inefficiencies, this lack of forethought can also lead to states from which it is impossible to recover and successfully complete a given task.

**Hybrid Architectures**

Although abstraction hierarchies were introduced to tame the exponential nature of deliberative search, they did not eliminate the lack of responsiveness because plans were still executed in an essentially open-loop fashion, even when interleaving planning with execution. More fundamentally, the exclusive use of symbols has been found to not be appropriate for the lower levels of control. However, symbolic representations are useful for planning that involves prior knowledge of the domain and task. As with higher level programming knowledge, the expectation is that a compiler or translator will efficiently reduce these abstract plans into directly executable low-level actions. Likewise, numerical computations are invaluable for low-level control, but are inappropriate when attempting to reason about processes and goals which are best described and understood at a higher level of abstraction, in much the same way that writing code in an assembly language is inappropriate for all but the lowest level of programming.

Hybrid architectures are a proposed solution to these complementary strengths and weaknesses of deliberative and reactive architectures that have been gaining increasing popularity among the robotics community [29]. As shown schematically in Figure 2-4, hybrid architectures consist of layers, each composed of different reasoning elements operating within various paradigms. At the top level is a deliberative planner which assimilates all available information and creates long-term global plans to be used by the lower levels. The lowest level consists of a behavior-based reactive planner which responds in real-time to sensory stimuli, but when possible also takes into account the higher level considerations or constraints passed down to it from above. The nature of this input from higher levels is discussed further in Section 2.1.4. In many hybrid architectures, an intermediate level also appears between the high level symbolic reasoning of the deliberative planner and the low level numerical computations of the reactive planner operating at the actuator control level [23].
2.1.2 Centralized vs. Distributed Processing

As with any complex system, tradeoffs must be made between the coherence, correctness, and straightforwardness of a centralized system on the one hand and the responsiveness, robustness, and flexibility of a distributed system on the other.

Centralized Architectures

The earliest mobile robot systems operated by gathering all available sensory data, creating a complete model of its static environment, planning an optimal series of actions within the context of that model, and then executing that plan [20] [47] [49]. The robot would then stop to gather more information and the process would repeat, as illustrated in Figure 2-5. Robotic systems of this sort were capable of moving about a room with simple static obstacles, essentially a “blocks world”, in an optimal manner; however, due to their inherent bottlenecks, problems arose with the use of centralized architectures when applied to more challenging environments. In such architectures, all sensor data must be collected and fused into a single monolithic world model, and a complete path considering all relevant information must be planned within this model before any single action can be taken, no matter how simple or urgent that action may be. Such systems contain no parallelism and no pipelining; once a plan has been formulated, it is blindly followed in an open-loop fashion, regardless of whether the actual environment differs from the model, and regardless of whether or not the commanded actions actually have the desired effect. In addition to introducing potentially harmful delays, a centralized architecture also leads to brittleness because the system may fail entirely if any single part of it is not functioning properly; thus, any modifications or additions made to a system function could cause a catastrophic error. In order to overcome the limitations inherent in a purely centralized architecture, schemes for allowing distributed processing to occur were developed.

Figure 2-4: Hybrid control architecture.
Hierarchical architectures are a type of distributed system in which the modules are organized into multiple control levels which operate at varying granularities, levels of abstraction, and time scales, thus providing varying tradeoffs between long-term correctness and completeness and short-term survival and relevance. Traditional hierarchical architectures use a homogeneous functional decomposition, with each level constructed of the same modules, albeit at different levels of reasoning [2]. They are composed of multiple control levels, each of which have the same type of structure as a centralized system, as shown in Figure 2-6. However, each level operates in parallel at a different rate, so that the lowest levels are free to respond to immediate stimuli without having to wait for higher level reasoning processes.

Another class of hierarchical architectures decompose the task itself in a recursive manner, following the abstraction hierarchy shown in Figure 2-3. A plan to achieve the highest level task is created and subtasks are recursively spawned to achieve subgoals of that plan until primitive actions achieve the desired result [22] [64], as shown in Figure 2-7.
While these hierarchical frameworks effectively bypass the sequential bottlenecks of purely centralized systems, their recursive decomposition imposes a rigid structure which has been found in practice to be overly constraining because whatever structure is chosen, it is not appropriate at all levels.

**Behavior-Based Architectures**

Behavior or schema based architectures constitute a radically different class of robot control systems. Rather than constructing the system of functional modules such as perception and planning, the system is composed of individual behaviors that each perform a specific, limited task. In principle, this allows the robotic system to be developed in a bottom-up, evolutionary manner because behaviors are essentially self-contained and capable of achieving their specific purpose whether or not other particular modules have been developed and are present in the system. A behavior encapsulates the perception, planning and task execution capabilities necessary to achieve one specific aspect of robot control. Thus, in such an architecture, not only the processing, but the actual control of the robot itself is distributed across multiple independent modules. The overall functioning of the system is defined not by a complex internal structure but rather by the interaction between simple behaviors and a complex world [67]. A behavior-based architecture is more robust than other types of architectures because if any one part of the system fails, the rest of the system continues to function independently. The drawback of such a system is that predicting its behavior is problematic; the interactions between large numbers of asynchronous independent modules are difficult to analyze and debug.
Because an individual component in a behavior-based architecture is capable of producing meaningful action, behaviors can be composed to form levels of competence [13], each of which endows the robotic system with a particular externally manifested capability, as shown in Figure 2-8. In theory, successive levels can be incrementally added in order to enhance the functionality of the robot without disrupting previously established capabilities; unfortunately, this is not always the case in practice. Such architectures most often use a priority-based arbitration scheme that selects the output of a single module and sends its commands to the vehicle controller [13] [60]; the output of other modules, and thus the knowledge contained within them, is lost. See Section 3.1, “Priority-Based Command Arbitration”, for a more complete discussion of this issue.

![Figure 2-8: Levels of competence in a behavior-based architecture.](image)

### 2.1.3 Sensor Fusion vs. Command Arbitration

Architectures can also be characterized by the ways in which they combine information and objectives. Architectures that perform sensor fusion construct a single unified world model based on all available sensory data [20] [48] [62]. This world model is then used for planning actions, as illustrated in Figure 2-9. This approach has the advantage of being able to combine evidence to overcome ambiguities and noise inherent in the sensing process [48], but has the disadvantage of creating a computationally expensive sensory bottleneck: all sensor data must be collected and integrated before it can be acted upon. A single monolithic world model is also more difficult to develop, maintain, and extend.

Another difficulty with sensor fusion is that information from disparate sources such as maps, sonar, and video, are generally not amenable to combination within a single representational framework that is suitable for planning such dissimilar tasks as following roads and avoiding obstacles. For example, ALVINN [57] uses an artificial neural network to associate video images of roads with appropriate steering directions and has been one of the most successful road following systems to date, yet it has been less successful than other systems such as Smarty [39] which use range data for the purpose of obstacle avoidance. Thus, by requiring a single representation for all sensor and map data, a centralized architecture does not allow specialized modules to use other representations and algorithms best suited to the task at hand.
In contrast, behavior-based architectures do not create a central world model; instead, the perceptual processing is distributed across multiple independent modules. Each behavior requires only fragmentary knowledge of the world and receives exclusively that sensory data which is directly relevant to its particular decision-making needs, thus each behavior is free to use whichever representation is deemed most appropriate for the purpose, and there is no need to fuse all available data into a single coherent world model before any individual module can respond to the processed data.

As described in Section 2.1.2, a behavior is a self-contained module which maps input data into actions as output, but that data and that action are only concerned with one specific aspect of controlling robot to achieve its tasks. Therefore, in a behavior-based system, it is necessary to select among or combine the actions suggested by the various behaviors to produce an action that meets the needs of the overall system; this is the role of the Command Arbitration module shown in Figure 2-10. By appropriately combining behavior commands through arbitration, a robot control system can respond to its environment without suffering the problems inherent in sensor fusion such as bottlenecks; however, command arbitration runs the risk of losing information valuable to the decision-making process. Therefore a careful balance must be struck between completeness and optimality on the one hand versus modularity and efficiency on the other.

Figure 2-9: Sensor fusion creates a centralized world model for planning.

Figure 2-10: Command arbitration combines behavior outputs to produce action.
2.1.4 Top-down vs. Bottom-up Control

There is general agreement in the robotics community that a layered architecture, as described in Section 2.1.1, is a highly useful, if not necessary, means of organizing systems for robot control. However, the nature of these levels and of the relationships between them is still vigorously disputed.

Some hierarchical architectures use a homogeneous functional decomposition, with each level constructed of the same modules, albeit at different levels of reasoning [2]; other hierarchical architectures decompose the task itself in a recursive manner, with higher levels selectively enabling and disabling modules at lower levels [22] [64]. In strictly hierarchical architectures, the design and operation of the robot control system are expected to proceed in a top-down manner; each level controls the level beneath it and assumes that its commands will execute as anticipated. Since expectations are not always met, there is a need to monitor the progress of desired actions and to report failures as they occur [66]; consequently, there exist the monitor processes attached to actions as seen in Figure 2-7. In an unstructured, unknown, or dynamic environment, this approach introduces complexities and inefficiencies which could be avoided if higher level modules participated in the decision-making process without assuming that their commands will be strictly followed [54]. In addition, only one module in a given level is activated by the level above and allowed to participate in the decision-making process, and a decision made at a higher level severely constrains the range of possible solutions which may be found by lower levels, often without sufficient information to warrant such a restriction.

These restrictions are eased by systems such as the Hughes architecture [52], which selects multiple behaviors for activation from a behavior pool, rather than a single behavior being active, as shown in Figure 2-11. Such architectures consist of a behavior-based subsystem which is exclusively responsible for control of the vehicle, and an event-driven higher level module that selectively enables and disables groups of behaviors at the lower level according to the currently appropriate mode of operation, thus allowing several behaviors to operate at once rather than predetermining a single active subtask module.

![Figure 2-11: Behaviors in Hughes architecture selectively activated from pool.](Figure reproduced from [52], with permission.)
In contrast to hierarchical architectures, all behaviors in the Subsumption Architecture are active at all times, in the sense that they always process data as it is available [13]. The control structure is bottom-up because each behavior determines for itself whether or not it is relevant to the current situation, based on the data that it receives. For example, a behavior designed to detect and avoid obstacles based upon sonar data would continually read the data to determine if there are obstacles in the robot’s path; when obstacles are detected the behavior would issue avoidance commands, otherwise it would remain quiescent. Hard-wired inhibition links are used to decide which behavior’s output is ultimately used to control the vehicle, as discussed in Section 3.1, “Priority-Based Command Arbitration,” on page 33. The hierarchical and behavior-based approaches are combined in the supervenience hierarchy approach, in which a partial order exists among behaviors, and a higher level behavior may only receive input from and issue commands to lower level behaviors [68].

Another class of bottom-up control structure is the emergent architecture, in which behaviors receive activation and inhibition from other behaviors, all of whom are competing for control of the robot. Both hierarchical structures and pre-defined prioritizations are avoided to allow the greatest possible reactivity to the current situation. In the emergent architecture described in [44], a behavior receives activation based on the current situation as with any other bottom-up mechanism, but via motivations, a behavior also receives activation from those current goals to which it contributes, as well as inhibitions from achieved goals with which the behavior would interfere. Spread of activation occurs forward from achieved effects and backward from desired effects through successor and predecessor links, respectively. The competition amongst behaviors arises by way of conflict links between behaviors that would interfere with each other. Such an architecture is more flexible and reactive than a pre-programmed structure, but has the disadvantage of being less predictable [43].

Another non-hierarchical behavior-based approach is to have high-level planners participate in the shared control of the robot, providing advice to the system rather than a plan to be executed. For example, systems built using the Hughes architecture [53] were successfully used for mobile robot control in which behaviors co-existed at various levels of reasoning and scales of time and space [54]. This structure, referred to as a free flow hierarchy, has been shown in simulation to perform action selection in a more robust manner [75].

### 2.2 Architectural Solutions

No one architecture can provide the best solution to all problems. Therefore, rather than assuming a dogmatic position that one class of architecture is better than another for all problems, the interrelated trade-offs discussed above must instead be resolved with a particular domain and application in mind, using aspects of various approaches as warranted. Any system must be capable of exhibiting purposeful behavior towards goals, whether or not those goals are explicit. For mobile robot systems, additional important considerations are that it must operate in real-time, provide fault tolerance, and always preserve the safety of the vehicle and of its surroundings.
One of the key characteristics of most mobile robot domains is that uncertainty plays a large role, and that is the class of domains being studied in this work. Prior knowledge of the environment may be incomplete or not exist at all, and the environment may be dynamic, so that reasoning must be non-monotonic in nature and occur rapidly enough to be able to respond to unexpected events. Posterior knowledge gained via sensing is incomplete, inaccurate, and uncertain, as is knowledge of the effects of actions decided upon and taken by the system. In addition, the dynamics of the vehicle itself often play an important role in determining which actions may be achieved and which actions are to be avoided.

Another important aspect of mobile robot systems is the need to combine information from several different sources such as video cameras, laser range finders, sonars, and inertial navigation systems; they must also be capable of combining objectives for a system that is to perform diverse tasks such as following roads, driving off-road, following designated paths, avoiding obstacles, and reaching goal destinations, as well as allowing for teleoperation. For a large, complex system, it is not reasonable to assume that all components will be constructed from scratch, adhering to strict guidelines of how they may perform their function. To be useful, an architecture must be readily usable by different programmers at different sites using different approaches. Therefore, the architecture should be able to accommodate and integrate subsystems that have been developed independently, so it must impose minimal restrictions on the nature of the data, representations, and algorithms used by these subsystems designed to achieve their respective tasks. Also, sensors operate at different rates, as do the procedures that process their data, so they must be allowed to operate asynchronously to maximize the throughput and thus the responsiveness of the system. While many general purpose architectures are computationally equivalent or nearly so, the ease with which various classes of systems may be instantiated within them varies greatly. A crucial yet often neglected consideration in the design of a mobile robot architecture is the ease with which a system may be developed, tested, debugged, and understood, as well as adapted to novel domains and applications.

DAMN is implemented as a group of distributed behaviors communicating with a centralized command arbiter, each behavior sending votes in favor of actions that satisfy its objectives and against those actions which do not. The arbiter is then responsible for combining the behaviors’ votes and generating actions which reflects their objectives and priorities. As such, DAMN implements the thesis that centralized arbitration of votes from distributed, independent, asynchronous decision-making processes provides coherent, rational, goal-directed behavior while preserving real-time responsiveness, and that a framework for developing and integrating independent decision-making modules communicating with such arbiters facilitates development and leads to evolutionary creation of robust systems.

The following are discussions of some of the design trade-offs made within DAMN to facilitate this desired symbiosis of purposefulness, comprehensibility, extensibility, flexibility, robustness, and reactivity.
2.2.1 *Deliberative vs. Reactive Reasoning*

Deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other’s deficiencies, allowing for purposeful goal-oriented behavior while retaining the ability to respond to potentially dangerous situations in real-time.

DAMN is a heterogeneous behavior-based system; each module uses those representations and planning paradigms deemed most suitable for the task for which it is responsible. Although DAMN is a behavior-based architecture, one important distinction between this system and purely reactive systems is that, while an attempt is made to keep the perception and planning components of a behavior as simple as possible without sacrificing dependability, they can and often do maintain arbitrarily complex internal representations of the world. Similarly, since DAMN behaviors are not constrained in what processing takes places within them, a behavior may be do as little or as much deliberation as is appropriate for its role in the system. Thus, as described in Chapter 4, behaviors range from simple proprioceptive responses to sophisticated map-based reasoning. Since both deliberative and reflexive modules are needed, DAMN is designed so that behaviors can issue votes at any rate; for example, one behavior may operate reflexively at 10 Hz, another may maintain some local information and operate at 1 Hz, while yet another module may plan optimal paths in a global map and issue votes at a rate of 0.1 Hz.

Each behavior is allowed to perform as much or as little explicit search as it deems appropriate; the only constraint is that the results of a behavior’s reasoning be expressed to the arbiter at a common level, as shown in Figure 2-12. For example, if the arbiter is receiving votes on which action to perform next, a reactive behavior may perform no search but simply vote on its the desirability of its chosen action relative to other possible actions, a deliberative behavior may search all the way until the final goal and then simply vote for the next action in its constructed plan, and any point in between may be undertaken by a behavior which performs lookahead until a certain point to determine which is the best next action which leads to the best result at the lookahead depth.

While the arbiters in DAMN use information from behaviors that are at any degree of reactivity or deliberation, the arbiters themselves may perform lookahead to produce action that responds to immediate circumstances and possibilities and yet considers the consequences of its actions in terms of predicted future states. In this way DAMN is similar to the approaches that interleave planning and execution, but rather than performing the lookahead up to the limits of available information, DAMN performs just enough lookahead to determine the next immediate action. Thus, DAMN is more toward the reactive end of the spectrum, yet avoids the pitfalls of a short-sighted reactive system that is effectively a using greedy search algorithm to perform gradient descent.
Figure 2-12: Combining deliberative and reactive reasoning into one command space.

DAMN only considers actions which are actually physically realizable at the moment, given the current state of the vehicle. The decision-making process repeats at a rate of several times per second, constantly re-evaluating what the current best action is. The results of the previous decision cycle are taken into account in the current decision process, thus providing some measure of persistence and hysteresis. Although this more local approach may produce less efficient actions than one which uses all available information, it allows a tighter loop between sensing and action and thereby avoids open-loop plan execution.

By performing lookahead, taking previous and future states into account, and allowing behaviors to perform at any level of reasoning, DAMN is able to produce rational actions to achieve goals; and by constantly evaluating the current situations and selecting among those actions which are available to the system at that moment, DAMN is able to maintain the responsiveness necessary for dealing with dynamic and uncertain environments.

2.2.2 Centralized vs. Distributed Processing

Centralized architectures provide the ability to coordinate, in a coherent fashion, multiple goals and constraints within a complex environment; however, a purely centralized architecture is clearly not appropriate for a real-time system where the environment is dynamic or uncertain. In addition to the bottlenecks which inhibit responsiveness, centralized architectures also suffer from the drawback that they are more subject to catastrophic failure than a distributed system; if any one portion of the system breaks, the entire system ceases to function because of their lack of modularity, centralized architectures are also more difficult to develop and maintain, and their reusability is severely limited.

Decentralized architectures offer the advantages of reactivity, flexibility, and robustness. However, a completely distributed system also has its disadvantages; in general, the behavior
produced does not reflect the multiple objectives and constraints that the system is subject to at any given moment, thus leading to significantly suboptimal performance. The sequence of actions produced may result in an incomplete plan that does not find a solution even when one does exists. Additionally, the interactions, both between modules and between the system and its environment, are less predictable and more difficult to understand and modify.

An issue which must also be addressed in distributed systems is the nature of the communications between modules. Direct communications between modules provides the system designer with a high degree of control over the operation of the system, which may be desirable when modules are engineered to interact in a very narrowly defined manner; however, this makes extensions and modifications to the system difficult, and greatly limits the extent to which the system may be reused in different domains (e.g. [13] [62]). Indirect communications via media such as a blackboard or a broadcast mechanism provides a layer of abstraction in the inter-module interface, thus simplifying the task of interchanging modules or adding new ones to a system; this flexibility comes at the cost of increased overhead, resulting in reduced efficiency and throughput (e.g. [27], [52]). Another form of indirect communication is those cases where no information at all is transmitted within the system from one behavior to another. In \textit{command fusion} architectures such as Motor Schemas [6], Fuzzy Control [30], and DAMN, each behavior communicates its intentions with a central arbiter or coordinator rather than with each other, thus allowing development and execution of each module to proceed completely independently; however, the central module may represent a bottleneck, and there may be benefits derived from mutually shared information that cannot be realized without inter-module communication.

The DAMN architecture reflects the position that some centralization is needed, but an appropriate level must be chosen so that it does not create a bottleneck. The capabilities of an agent should be divided up as finely as is practical among task-achieving behaviors, and the interfaces must be defined so as to avoid being overly restrictive. Thus, as shown in Figure 2-13, independent behaviors operate in a distributed fashion to generate votes for actions based on domain-specific knowledge, while a central arbiter combines their results to generate reasonable behavior that satisfies as many objectives as possible.

![Figure 2-13](image) Centralized arbitration of votes from distributed behaviors in DAMN.
The centralized arbiter provides coherent, rational, goal-directed behavior while the distributed behaviors preserve real-time responsiveness to the environment. In order to preserve the respective advantages of centralized and distributed architectures, sufficient information must be communicated from the behaviors to allow the arbiter to make intelligent decisions, but the arbiter must not be so complex as to become a bottleneck for the system. Various points along this trade-off spectrum have been explored within DAMN, using different types of arbiters and vote structures.

Because the behaviors in DAMN are completely modular and independent, new capabilities may be added to an existent system without a need to disrupt or modify previously established functionality. Additionally, because the behaviors operate asynchronously, each at their rate without the need to synchronize with a central clock, throughput and reactivity is maximized. Thus DAMN provides a framework which facilitates the evolutionary development and integration of independent decision-making modules to create robust systems of incrementally greater capabilities.

### 2.2.3 Sensor Fusion vs. Command Arbitration

Sensor fusion is useful in overcoming ambiguities and inconsistencies in data, but in order to preserve responsiveness, it must not be a central component through which information must pass before action can be taken. In addition, it may be desirable to use more than a single approach to various aspects of perception, planning and control. Command arbitration avoids the sensor fusion bottleneck, but risks making suboptimal decisions when multiple factors must be simultaneously considered.

Within DAMN, arbiters perform command arbitration to obviate perceptual bottlenecks, and the behaviors are defined to be completely independent, so that each behavior may use whichever representation and algorithm is most appropriate for that specific task for which it is responsible. Behaviors provide a total encapsulation of domain and procedural knowledge whose details no other module need be concerned with, so that, as in the example in Section 2.1.3, a road following behavior may be implemented as an artificial neural network processing video images while obstacle avoidance may be implemented by constructing a local map from range images.

Unlike other behavior-based systems that use priorities to effect a traded control system, DAMN takes a shared control approach where several modules concurrently have some responsibility for control of the robot. Input from all behaviors are used in determining what the next action should be, so that there are multiple influences on the decision-making process, and compromises are made to satisfy as many objectives as possible.

The issues involved in action selection and the trade-offs involved are explored in detail in Chapter 3, along with the approach taken by the four different arbitration schemes implemented in DAMN.
### 2.2.4 Top-down vs. Bottom-up Control

Rather than imposing an hierarchical structure or using a prioritized behavior-based system to effect a traded control system, DAMN takes a shared control approach where several modules concurrently have some responsibility for control of the robot. In order to achieve this, a common interface is established so that modules can communicate their intentions without regard for the level of planning involved. The distinction made in DAMN is not in the level of abstraction of a given module, but rather whether its domain is represented and acted upon in a discrete or continuous manner. As shown in Figure 2-14, all continuous servo-like activity is instantiated as a voting behavior without regard for the time or space scale in which it operates; votes from behaviors operating at all levels of reasoning cooperate in determining what the next action should be. Sequential activity is controlled by a mode manager such as SAUSAGES [26], which exert meta-level control within DAMN by modifying the voting weights assigned to behaviors and thus controlling the degree to which each behavior may influence the system’s decision-making process and thus the robot’s actions.

![Figure 2-14](image.png)

**Figure 2-14**: Tradition “planning levels” function at the same level in DAMN.

DAMN functions at the level of geometrical reasoning, which is essential to the successful performance of a mobile robot system. All behaviors, whether a map-based global path planner or a reactive obstacle avoidance behavior, communicate in terms of a common language which specifies desired states of the system. In this way, DAMN serves as a bridge between the symbolic reasoning of high-level planners and the numerical calculation at the actuator control level, and does so without imposing the top-down constraints and assumptions of an hierarchical architecture.
2.3 Summary

The essential trade-offs in the design of a mobile robot control architecture considered in this chapter were centralized vs. distributed processing, reactive vs. deliberative reasoning, sensor fusion vs. command arbitration, and top-down vs. bottom-up control. The properties of some representative architectures along these four dimensions are summarized below in Table 1. While these four issues are interrelated and must be considered together, it can be seen in this table that they are at least partially separable.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Deliberative/Reactive</th>
<th>Centralized/Distributed</th>
<th>Fusion/Arbitration</th>
<th>Top-Down/Bottom-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shakey [49]</td>
<td>Deliberative</td>
<td>Centralized</td>
<td>Fusion</td>
<td>Top-Down</td>
</tr>
<tr>
<td>NASREM [2]</td>
<td>Both</td>
<td>Distributed</td>
<td>Fusion</td>
<td>Top-Down</td>
</tr>
<tr>
<td>TCA [64]</td>
<td>Both</td>
<td>Distributed</td>
<td>Fusion</td>
<td>Top-Down</td>
</tr>
<tr>
<td>Codger [24]</td>
<td>Both</td>
<td>Both</td>
<td>Fusion</td>
<td>Bottom-Up</td>
</tr>
<tr>
<td><strong>DAMN</strong></td>
<td>Both</td>
<td>Both</td>
<td>Arbitration</td>
<td><strong>Bottom-Up</strong></td>
</tr>
<tr>
<td>GAPPS [60]</td>
<td>Both</td>
<td>Centralized</td>
<td>Arbitration</td>
<td>Bottom-Up</td>
</tr>
<tr>
<td>Hughes [52]</td>
<td>Both</td>
<td>Distributed</td>
<td>Arbitration</td>
<td>Bottom-Up</td>
</tr>
<tr>
<td><strong>Subsumption</strong> [13]</td>
<td>Reactive</td>
<td>Distributed</td>
<td>Arbitration</td>
<td>Bottom-Up</td>
</tr>
</tbody>
</table>

Table 1: Characterizations of design trade-offs made in representative architectures.

On one extreme of the trade-offs is the architecture developed for control of Shakey the Robot [49]; it is deliberative, centralized, uses sensor fusion, and control is top-down. This is collectively known as a “centralized” architecture, but as can be seen, there are several variables that can be independently manipulated. At the opposite extreme is the prototypically “reactive” Subsumption Architecture [13]. In between are hybrid architectures which contain both deliberative and reactive elements and vary along the other three dimensions.

Generally speaking, an architecture must be distributed in order to have the property of being reactive; the exception to this rule is the GAPPS architecture [60], which achieves reactivity in a centralized system by compiling all decision making into a highly efficient combinatorial network. There is also a correlation between the extent to which an architecture is top-down and
the extent to which it is deliberative; committing significant resources to the construction of high level plans is only worthwhile if those plans will actually be have a significant impact on the control of the system, which is to say that it must be top-down and the environment must be known a priori with a great deal of precision and certainty. A top-down system must also perform sensor fusion to provide the input needed by the higher-level modules. Alternatively, a system will be able to react to new and unexpected circumstances with greater speed and opportunism if control proceeds in a bottom-up fashion; otherwise the system just continues to do what it had already planned to do.

DAMN has both centralized and distributed components and manages to preserve the most salient beneficial features of each. Blackboard systems such as Codger [24] are the only other type of architecture that also has both centralized and distributed components. However, unlike behavior-based systems, blackboard systems perform synchronization and sensor fusion, both of which reduce throughput and increase latency, thus greatly reducing their responsiveness. In addition, synchronous operation requires greater coordination between modules, both in their development and operation, thereby increasing the difficulty of integrating systems.

Thus, in accordance with the thesis presented in this document, DAMN performs centralized arbitration of votes from distributed, independent, asynchronous decision-making processes and in so doing provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment. DAMN also provides a framework for developing and integrating independent decision-making modules communicating with such arbiters, thus facilitating their development and leading to evolutionary creation of robust systems of incrementally greater capabilities.
CHAPTER 3: ACTION SELECTION

When designing a software architecture for the control of complex real-time systems such as mobile robots, there are many important issues that must be addressed. The architecture must provide the means by which the system may accomplish its multiple objectives efficiently; it must be able to satisfy real-time constraints, promote fault tolerance, and provide for the safety of the vehicle and of its surroundings. The high-level tradeoffs that must be considered when designing a control architecture and the decisions made in DAMN were discussed in Chapter 2; to be effective in dynamic, uncertain, and complex domains, DAMN combines both deliberative and reactive elements within a bottom-up structure, and centralized arbiters combine commands received from distributed, asynchronous behaviors.

Having decided upon a behavior-based architecture, it is then necessary to specify how information in a distributed system is to be combined and used to generate action in an intelligent, coherent manner. Systems that combine command inputs to determine an appropriate course of action fall into two broad categories, priority-based arbitration and command fusion; these are described in Section 3.1 and Section 3.2, respectively.

Based on experience with an architecture that employs priority-based arbitration [34] and the limitations encountered in practice [54], DAMN was devised to provide a general framework for command fusion that overcomes some of the limitations of earlier command fusion systems. The three arbitration schemes within DAMN that are forms of command fusion are described in Section 3.3 and compared with previous behavior-based architectures. Limitations that remain in command fusion and in the DAMN arbiters are then explored in Section 3.4, and Section 3.5 introduces a new form of action selection via utility fusion as a means of overcoming these limitations.

These various forms of action selection in DAMN represent points along the spectrum between purely decentralized and purely centralized architectures. The implementation details for the four types of arbiters are provided in Chapter 4.

3.1 Priority-Based Command Arbitration

In a distributed architecture, it is necessary to decide which behaviors should be controlling the vehicle at any given time. In some architectures, this is achieved by having priorities assigned to each behavior; of all the behaviors issuing commands, the one with the highest priority is in control and the rest are ignored. This allows for quick responses to new situations and stimuli, although the prioritization only allows one module to affect control at any given time.

Subsumption Architecture

The Subsumption Architecture [13] is perhaps the most well-known example of an architecture that employs priority-based command arbitration. In the Subsumption Architecture, the
prioritization is implicit in the wiring of behavior modules; a higher-level behavior can override the output of a lower-level behavior via an *inhibition* link, as shown in Figure 3-1a. While BEHAVIOR Y is dormant, the output of BEHAVIOR X is used to control the vehicle. When Y is active, its output inhibits the output of X and is used to control the vehicle instead. In a similar fashion, a higher-level behavior can overwrite the input of a lower-level behavior via a *suppression* link, as shown in Figure 3-1b. BEHAVIOR X operates as before, unaware of the source of its input; under normal circumstances the input source is the same as always, but when Y is active then its output is used as input to X instead. The Subsumption Architecture is designed to grow in an evolutionary manner; the higher level capability of BEHAVIOR Y is to be added to the system without disturbing the lower level capabilities already offered by X. These behaviors can thus be composed to form *levels of competence*, each of which endows the robotic system with a particular externally manifested capability, as discussed in the “Behavior-Based Architectures” subsection of Section 2.1.2.

![Figure 3-1](image)

*Figure 3-1*: Combining behaviors in the Subsumption Architecture. Behaviors interact via a) inhibition, or b) suppression.

**GAPPS**

In the GAPPS architecture [60], such priorities are compiled into a *mediator*, which combines inputs from the behaviors through a combinatorial logic circuit which yields the system output, as shown in Figure 3-2. This decomposition is recursive and hierarchical; each level may be composed of sub-levels sending their output to a mediator, whose output is in turn used as input to a higher level mediator. Thus, more complex arbitration is possible without loss of real-time performance. As in the Subsumption Architecture, behaviors within GAPPS are implemented as Finite State Automata whose operation and intermodule communication are highly constrained.

![Figure 3-2](image)

*Figure 3-2*: Combinatorial circuits for action selection in the GAPPS architecture.
Hughes Architecture

The lowest level of the hybrid Hughes architecture [52] also consists of a similar distributed control mechanism; however, a centralized blackboard [27] is used for arbitration and communication between behaviors whose priorities are explicit and can be changed dynamically, as shown in Figure 3-3. Unlike the Subsumption Architecture and GAPPS, the implementation of behaviors in the Hughes Architecture is not limited to a particular paradigm, and arbitrary communication between modules may occur via the blackboard. However, as with the Codger system [62], it was found in practice that modules have very narrow communication needs with other specific modules, and a general framework such as a blackboard introduces unnecessary latencies and bottlenecks, as well as increasing system complexity.

![Figure 3-3: Communication and arbitration among behaviors via a central blackboard.](Figure reproduced from [52], with permission.)

Emergent Architecture

In an emergent architecture [44], behaviors receive activation and inhibition from other behaviors and compete in a winner-take-all fashion to decide which behavior’s command is used to control the robot at any given moment. The behaviors also receive activation and inhibition based on the current state of the world and on the current goals of the system, effectively determining a dynamic prioritization of behaviors. Spread of activation occurs forward from achieved effects and backward from desired effects through successor and predecessor links, respectively. The competition amongst behaviors arises by way of conflicter links between behaviors that would interfere with each other. Such an architecture is more flexible and reactive than a pre-programmed structure, but has the disadvantage of being less predictable [43].

3.1.1 Limitations of Priority-Based Command Arbitration

As discussed in Section 2.1.2, “Centralized vs. Distributed Processing”, behavior-based architectures successfully avoided the sensing and planning bottlenecks of centralized systems, thus gaining the important advantage of reactivity. Additionally, these architectures introduced the
important notion of decomposing the system according to the tasks to be achieved, rather than the
traditional functional decomposition, along with the associated idea of incremental, evolutionary
system development. However, the objectives of independent behavior development and the
accumulative addition of new capabilities were not realized in practice. One of the requirements
for a robot control system is that it be capable of satisfying multiple, possibly conflicting goals
[67]; as the name “subsumption” implies, this is achieved in priority-based architectures by
having one behavior’s commands completely override another’s. While this is an effective
scheme for choosing among incompatible commands, it does not provide an adequate means for
dealing with multiple goals that can and should be satisfied simultaneously.

This inability to compromise stems from the fact that priority-based arbitration of commands
masks all of the knowledge within each individual behavior that was used to reach its decision. In
the process of selecting a command, behaviors may weigh several alternatives before making its
ultimate choice. In other behaviors, it may be more natural to merely rule out certain
inappropriate actions rather than selecting a desirable one, yet their only means of expression is to
completely suppress or inhibit the outputs of other behaviors. A compromise between behaviors
cannot be achieved in such an all-or-nothing scenario; whenever one behavior’s output is
inhibited by another, the information and knowledge represented by that behavior is completely
lost to the system.

Consider, for example, the evolutionary development of a system that is capable of avoiding
obstacles while following a road in scenarios like that suggested by Figure 3-4a. Adopting the
symbology of the Subsumption Architecture, the initial system would consist of a FOLLOW-ROAD
behavior as shown in Figure 3-4b. A second behavior, AVOID-OBSTACLES, is then developed to
provide the ability to detect and avoid obstacles; the obvious way to add it to the system is to
effectively give it a higher priority via an inhibition link, as shown in Figure 3-4c.

Since, in general, there are no obstacles to be avoided, AVOID-OBSTACLES is usually quiescent, so
that most of the time FOLLOW-ROAD controls the vehicle as it always has. Then, as the vehicle
approaches an obstacle, the AVOID-OBSTACLES behavior perceives it and begins to send obstacle
avoidance commands which inhibit the output of the FOLLOW-ROAD behavior, thus taking over
control of the vehicle until it has passed the obstacle, at which point FOLLOW-ROAD resumes
control. The difficulty with this approach arises when the AVOID-OBSTACLES behavior is active;
during that time, the output of the FOLLOW-ROAD behavior is ignored, and thus the
road-following capability of the system is temporarily disabled. As illustrated by the arrows in
Figure 3-4a, when an obstacle is in front of the vehicle, AVOID-OBSTACLES may issue commands
for the vehicle to turn either left or right to avoid the obstacle, but has no knowledge that it is
preferable to go left so as to stay on the road.
These types of problems were encountered by a robot control system that was developed using the prioritized behavior-based paradigm [34]. In this system, the limitations of priority-based arbitration were overcome by creating one larger behavior which received both road and obstacle data and internally combined the procedural knowledge of both the road-following and obstacle avoidance behaviors, as shown in Figure 3-4d. Another possibility, shown in Figure 3-4e, would be to have the newly added obstacle avoidance behavior also receive road data, combine the two representations, and suppress the input to the road following behavior with a new input that would effectively deceive the road-following behavior into taking the correct action. Although these techniques provide a means of controlling the vehicle so that it would stay on the road even when avoiding obstacles, such practices fundamentally violate the desired architectural features of modularity, independent development, distributed perception, and distributed control [58].

Another problem with one behavior completely inhibiting the output of another is that it undermines one of the purported advantages of behavior-based architectures: that a given level of competence, once established, can always be relied upon to imbue the system with certain capabilities, regardless of subsequent additions to the system. If the output from a lower level is inhibited, or if its input data is suppressed so that it is acting upon data for which it was not developed, then any assurance of performance and capability which that level previously provided can no longer be relied upon.
3.2 Command Fusion

A different means of action selection is to combine commands from various behaviors. Prioritization only allows one module to affect control at any given time; the output of other modules, and thus the knowledge contained within them, is lost. To overcome this limitation, some architectures perform *command fusion*, i.e., they combine the commands from individual behaviors so that decisions may be made based on multiple considerations while preserving the modularity and reactivity of distributed systems.

Schema theory has proven to be a useful and interesting framework for applying Distributed Artificial Intelligence techniques to the robotics domain, and is used here to describe the various means of command fusion that exist. Schemas are independent modules which represent perceptual information and contain processes for deciding how to act on that information; together, they constitute a distributed system for reasoning and for acting, either externally to change the state of the physical world or internally to change the state of the system.

Perceptual schema map sensed data into object representations which in turn are used by motor schema to generate actions. Unlike priority-based arbitration which selects the output of a single behavior, the outputs of all schema are combined in order to produce an action that reflects multiple considerations. This combination of schema is also prioritized, in that each schema has a weight associated with its output command that reflects the importance of the command or the certainty of the information on which it is based.

Arbib and House developed two different schema models to explain behaviors that were observed in frogs [5]. The Orientation Selection model chooses among orientations based on a summation of excitatory and inhibitory influences from each schema, and the Vector Field model of path planning associates motor actions with schema objects and combines them. In applying schema theory to the problem of a toad trying to reach a prey while avoiding obstacles, Arbib and House postulate that two separate depth maps are maintained for prey stimuli and obstacle stimuli, that they are individually processed to generate orientation preferences, and that the motor schemas combine these preferences in order to produce coordinated action.* Mobile robot architectures that perform command fusion are classified according to whether they adhere to the Orientation Selection or Vector Field model and their operation is described in that context.

3.2.1 Orientation Selection Methods

In the Orientation Selection (OS) model developed in [5], an obstacle-avoidance schema convolves the barrier depth map with an inhibitory mask and a goal-seeking schema convolves the goal depth map with an excitatory mask. The resulting arrays output by these two schemas are then summed to create a net excitation array which represents the combination of all excitatory

* A more complete description of the schema models and of the behavioral studies can be found in Appendix A, along with a comparison of the two schema paradigms in the context of explaining observed animal behavior, and implementations of these models for robotic control are compared for effectiveness.
and inhibitory influences. This excitation array is integrated over distance for several orientations to yield an histogram of total excitation as a function of orientation; the orientation corresponding to the highest value in that histogram is then selected as the direction in which to move. In this way, an action is decided upon by combining the influences of all available inputs.

Although not directly a result of this model, there are robotic systems in use that nonetheless are instantiations of this Orientation Selection Model; they evaluate paths along a depth map in various directions from the current robot position, and select that path which maximizes the summed excitatory and inhibitory influences.

**Hughes ALV System**

The system developed at the Hughes Research Labs for the navigation of the Autonomous Land Vehicle [18] functioned by first building a terrain elevation map from laser range sensor data (Figure 3-5a). A suspension model of the vehicle was then applied along several candidate linear paths to determine the existence and location of terrain which the vehicle could not safely traverse (Figure 3-5b). The distance which the vehicle could travel before encountering such an obstacle along each path was then used as a measure of the path’s desirability (Figure 3-5c). Each path was then averaged with its two immediate neighbors to provide some measure of smoothing, and the one with the highest value was then chosen. Although this system was behavior-based and the intent was for the orientation selection process to be fully distributed, the priority-based arbitration that occurred via a blackboard created the difficulties described in the previous section. As was shown in Figure 3-4d, the limitations of priority-based arbitration were overcome by creating one larger behavior which internally combined the functions of multiple behaviors, so that the system was no longer truly distributed.

**Figure 3-5**: Hughes ALV system.

- a) process range data to create elevation map;
- b) obstacle locations are determined by applying vehicle suspension model to map;
- c) traversable length of linear paths are determined.
**Ranger**

The RANGER system [35] uses many of the same techniques developed for the Hughes system; it too builds an elevation map from sensory information and determines the traversability of several candidate directions of travel, using a simpler function of the terrain rather than a vehicle suspension model. RANGER improves upon the Hughes system by using a much higher fidelity model of the vehicle and system dynamics to generate trajectories which the vehicle controller is capable of executing, rather than linear paths which cannot be executed faithfully. RANGER also uses adaptive filtering to improve system efficiency and guarantee response times. These changes allowed for navigation at speeds significantly higher than was previously possible. The RANGER system is specifically designed for and is limited to determining terrain navigability from range images; it has been integrated as a behavior within DAMN to create systems with more diverse capabilities.

**Vector Field Histogram**

The Vector Field Histogram [11] is another method used for robot control which follows the Orientation Selection model, although it is not a distributed system. The Vector Field Histogram (VFH) method begins by building an egocentric map of observed obstacles. It does so by creating a local histogram grid whose elements represent the number of times the robot’s sensors have reported the presence of an obstacle at that location, as illustrated in Figure 3-6a. This corresponds to the input obstacle depth map used in the OS model, with the addition of the histogram counts which provide the certainty information which was hypothesized but not included in the OS model. The next stage of processing involves integration along various orientations to yield a one-dimensional polar histogram of obstacle density (Figure 3-6b).

The histogram is then smoothed so that orientations adjacent to obstacle directions also receive inhibitory influence. Finally, a threshold \( \tau \) is applied to the polar histogram, and those areas whose obstacle density is below that threshold are considered as candidate directions for movement. The presence of a goal location is now combined, not by summation, but by selecting the candidate direction which is closest to the direction toward the goal. While the Vector Field Histogram method has been an effective navigation system, it is somewhat limited in that it only deals with obstacle avoidance while pursuing a single goal location.

![Figure 3-6 : Vector Field Histogram.](image)

a) obstacle grid with detection counts, b) thresholded orientation histogram
Fuzzy Logic

Many control systems have been constructed using fuzzy logic [42], including a fuzzy control system developed for mobile robot navigation [30]. Fuzzy logic defines a mathematics in which set membership is not binary; an object has a degree of membership in a set which lies in the real numbered interval between 0 and 1, inclusive. For example, Figure 3-7a shows the fuzzy membership sets for left, straight, and right; a point at some orientation $\theta$ on the x-axis belongs to each of the three sets to the extent indicated by their functions. The black circle indicates the presence of an obstacle at that orientation, so it belongs to the set left with value 0.8, and the sets straight and right with value 0.0.

![Figure 3-7: Command fusion via fuzzy logic.](image)

a) fuzzy membership sets, b) output of rule to turn right when an obstacle is on the left, c) output of several fuzzy rules, d) sum of all rule outputs and defuzzification strategies, e) center of mass defuzzification yields averaged commands.
If-then rules take these fuzzy variables as antecedents and produce fuzzy inferences as a result [78]. For example, there may be a rule of the form “If there is an obstacle on the left, then turn right” with an associated fuzzy multiplier of 0.75, which produces the fuzzy output shown in Figure 3-7b. Figure 3-7c shows the output of several such rules, which are then summed to produce the result in Figure 3-7d. Various defuzzification strategies such as selecting the maximum value or the center of mass to select a single crisp output to be sent to the controller.

The DAMN actuation arbiter has also been recast into a fuzzy logic framework and used for control of a mobile robot [77]. However, as we shall see, DAMN is more general in that it allows for an arbitrary distribution of votes rather than being restricted to take on the shape of a particular function. In addition, DAMN is not rule-based but rather behaviors evaluate input and produce votes in the manner deemed most appropriate to the task. The DAMN arbiter output selection process is similar to defuzzification via maximum value; other commonly used defuzzification schemes such as center of mass are not appropriate for mobile robot control because they effectively average commands, which can produce results which are poor at satisfying any of the goals of the system, as in the case shown in Figure 3-7e.

### 3.2.2 Vector Field Methods

In the Orientation Selection model, a level of excitation was determined for each point in a local depth map; the Vector Field model seeks to associate with each point not a scalar value, but rather a vector indicating which direction the agent should head in were it to pass through that point. The magnitude of the vector indicates the strength of the choice for moving in that particular direction, i.e., a small vector at a point means that heading in a different direction would have a relatively small effect, while a large vector indicates that the choice of orientation at that position is an important one. Thus, a goal-seeking schema sets up an attractive field of vectors oriented towards the goal, as shown in Figure 3-8a, and an obstacle-avoidance schema associates with each obstacle a repulsive field like that shown in Figure 3-8b. A row of obstacles, such as a series of fence posts, are combined to create the vector field in Figure 3-8c, which would tend to move an object towards either end of the fence. The agent also has a vector field associated with itself which serve to propel itself forward, as shown in Figure 3-8d. For each of these types of fields, the strength of the vectors diminishes with distance from the object creating the field. The schemas are then combined by summing their associated vector fields, as in Figure 3-8e, which shows the vector field resulting from the combination of schemas representing the agent facing a goal that lies behind a fence. Integration of these vector fields to generate a path would yield two “bundles” of trajectories, one leading around the fence to the left and the other to the right.

**Potential Fields**

Potential Fields is a method of planning robot trajectories based on combining the vector fields induced by mapped objects, such as repulsive fields from obstacles and attractive fields from goals [8] [37]. This approach to mobile robot control is very similar to the Vector Field model; as such, it has become the control paradigm most closely associated with motor schema theory.
Figure 3-8: Component and combined vector fields.

a) attractive toward goal; b) repulsive from obstacle; c) repulsive from obstacle barrier;
   d) self-induced forward propulsion; e) combined fields from a, c, and d;
   f) combined potential fields for obstacle and goal schemas, and resulting path.

(Figures a-e reproduced from [4], f reproduced from [7], with permission.)
Work on motor schemas has been directly inspired by the Vector Field model and uses potential fields to implement mobile robot analogs of the prey-attractant and barrier-repellent schemas, as well as motor schemas such as stay-on-path and avoid-moving-obstacle [7].

In the Vector Field model, the entire field is created at once, and then a path is planned through it and followed to the goal. The result of using these combined fields to plan a path for a mobile robot is shown in Figure 3-8f. This would be advantageous if the obstacle and goal locations were known with certainty \textit{a priori}; in practice, however, it has been found that it is necessary for a robot to reevaluate its path often in order to maintain reactiveness in a dynamic and uncertain environment. In the implementations of Potential Fields for robot control, only the force vector for the current robot position is calculated for each schema. The sum of these vectors is then used to control the instantaneous motion of the robot, as shown in Figure 3-9. The process is then repeated from the next robot position to be evaluated, taking into consideration any new perceptual information that may have become available.

![Figure 3-9](image_url)

\textbf{Figure 3-9 :} Vector addition from potential field motor schemas.

The Motor Schema framework [6] allows multiple behaviors to be instantiated and their outputs combined for a variety of different tasks and environments, and is therefore more general than the OS systems that only select a direction based on obstacle and goal locations. However, it suffers from the well-known problem of local minima in potential fields, although there are various means of overcoming this limitation. Another, perhaps more serious problem, is that arbitration via vector addition can result in an averaged command which is not satisfactory to any of the contributing behaviors; for example, a robot cannot pass through closely spaced obstacles as in Figure 3-8c. This is problematic for robots that wish to explore areas that may only be accessible by passing through narrow entrances such as doorways. Borenstein also reported conditions under which potential fields suffer from oscillations and instability [12]. Potential fields have the advantage that the entire path may be computed if the obstacle and goal locations are known \textit{a priori}; however, it has been found in practice that it is necessary for a robot to reevaluate its path often in order to maintain reactiveness in a dynamic and uncertain environment.
3.3 DAMN Command Arbitration Methods

Various methods of command arbitration have been explored within the DAMN framework, using different types of arbiters and vote structures. Mainly due to real-time concerns, sensor fusion is avoided as an integral part of the system, although, when appropriate, it may be used within behaviors to overcome ambiguities and inconsistencies in data. However, architectures that perform command arbitration make suboptimal decisions and, as discussed above, are particularly problematic when multiple factors must be simultaneously considered.

Unlike behavior-based systems that use priorities to effect a traded control system, DAMN takes a shared control approach where several modules concurrently have some responsibility for control of the robot. Input from all behaviors are used in determining what the next action should be, so that there are multiple influences on the decision-making process, and compromises are made to satisfy as many objectives as possible. However, no averaging of commands takes place as in the fuzzy logic or potential fields methods, but rather a command is chosen that maximizes some function of the behaviors’ inputs. There are currently four different arbitration schemes in use in DAMN, each with their own advantages and disadvantages. The three schemes which are forms of command arbitration are described below and compared with previous architectures that perform command arbitration; the limitations in these DAMN arbiters are then discussed in the section that follows. The fourth scheme, which introduces a new means of action selection based on utility fusion to overcome these limitations is described in Section 3.4. Implementation details for the four arbiters are provided in Chapter 4.

3.3.1 Constraint Arbitration

The simplest DAMN arbitration scheme is one in which each behavior sends to the arbiter the highest command value that satisfies the constraint which that behavior seeks to enforce. The arbiter then takes the minimum of these values, thereby simultaneously satisfying all constraints. For example, speed behaviors determine their maximum speed vote based on constraints such as avoiding vehicle tip-over and avoiding obstacles, and the arbiter selects the slowest of these speed votes, as shown in Figure 3-10.

![Figure 3-10: Arbitration of maximum speed constraints.](image)

The DAMN constraint arbitration scheme can be cast into a priority-based framework. For example, the speed arbiter effectively assigns priorities dynamically based on the value of the speed constraint; the behavior with the slowest speed command has the highest priority, and its
value is used to control vehicle speed. This provides a very simple form of speed control that is adequate to the systems that use it, but would be inadequate to allow for the inclusion of more sophisticated considerations which cannot be specified as a single-valued upper bound. The limitations of priority-based arbitration render this type of arbitration ineffective when multiple speed objectives, such as safety vs. timeliness, must be balanced against each other and compromises are to be made.

### 3.3.2 Actuation Arbitration

In the second DAMN arbitration scheme, each behavior votes for or against various alternatives in the actuator command space. For example, the turn arbiter receives votes for a fixed set of vehicle curvatures. Each behavior generates a vote between -1 and +1 for every possible steering command, with negative votes being against and positive votes for a particular command option. Each behavior also has an associated weight which reflects its priority relative to other behaviors. The arbiter then computes a weighted sum of the votes received for each steering command, and selects that action which has the highest total score. In the example illustrated in Figure 3-11, two behaviors are active, one responsible for obstacle avoidance and the other for goal seeking (only five turn options are shown for simplicity of exposition). The magnitude of a vote is indicated by the size of a circle, with a large unfilled circle representing a vote of +1, a large striped circle a value of -1, and a small circle a value near 0. Thus, the goal-seeking behavior is voting most strongly in favor proceeding straight and less favorably for a soft left turn, and voting against hard left or any right turns; the obstacle avoidance behavior is voting against a hard left or soft right, and allowing the other turns as acceptable, with soft left being the most favorable.

Because avoiding obstacles is more important than taking the shortest path to the goal, the obstacle avoidance behavior is assigned a higher weight than the goal seeking behavior, as indicated by the thicker arrows in the diagram. The arbiter then computes a weighted sum of the votes it has received from each behavior, and the command choice with the highest value is selected and issued to the vehicle controller. In this case a soft left turn would be executed, since its weighted sum is the greatest, thus avoiding any obstacles while still more or less moving toward the goal.

![Figure 3-11](image)

**Figure 3-11**: Turn behaviors vote for candidate vehicle curvature commands. Circle size indicates vote magnitude from 0 to 1; striped circles represent negative values.
This class of DAMN arbiters follows the Orientation Selection model described in Section 3.2.1. In the arc-based turn arbiter, each behavior votes for or against each of a set of possible orientations; Figure 3-12 (a&b) shows the votes from two behaviors plotted with vote value $\nu$ as a function of curvature $\kappa$. The arbiter then combines these votes as a weighted sum, shown in Figure 3-12c, using assigned behavior weights of 0.8 and 0.2, respectively. This is essentially the same process as the integration of excitation in the OS model, and it too allows excitation and inhibition based on various inputs such as goals and obstacles. The resulting histogram is then smoothed, which again serves to laterally spread excitation along the orientation axis, the orientation with the highest value is selected as the direction to move in, and finally sub-pixel interpolation is performed to overcome the effects of discretization, as shown in Figure 3-12d. This process is described in greater detail in Section 4.1.1.

This type of DAMN arbitration is very similar to Fuzzy Logic systems, and in fact has been recast into this framework [77]. However, the behaviors are not necessarily using fuzzy membership explicitly, but instead may be independently evaluating each possible action and collecting the results into a distribution which may take on any shape. After the arbiter smooths the combined inputs, it defuzzifies using the “maximum” criterion. Other defuzzification strategies such as center of mass assume a unimodal function, and in the general case an averaging of inputs would select inappropriate commands. For example, the weighted sum in Figure 3-12c has two areas of positive votes, one for soft left turns and another for hard right; selecting the center of mass would result in a soft right turn, which has negative vote sums and would have bad effects such as driving the vehicle directly into an obstacle.

This type of actuation arbitration is very general and many different types of behaviors for various tasks have been readily integrated within it, including the Ranger and Hughes systems for off-road navigation described above, as well as Smarty [39]. Another orientation selection behavior included within DAMN is Ganesha [40], which creates a local two-dimensional histogram of sensed obstacles and evaluates arcs within this map. The Vector Field Histogram could easily be included as a DAMN behavior in the same manner without a need for thresholding, and would thus be capable of combination not only with a goal direction but with any other arbitrary task.

This arbitration scheme provides a means by which commands can be combined, unlike action selection schemes that choose a single behavior’s command to be used in controlling the robot. However, the information supplied to the arbiter is somewhat minimal so that it is unable to take into consideration the dynamics of the plant being controlled, i.e., the vehicle’s speed and turn radius; it is assumed that behaviors will be able to update their votes at a sufficiently fast rate compared to vehicle speed, and that those votes will be acted upon quickly enough by the system, that dynamics do not need to be considered. In addition, it is assumed that a new set of votes is received from a behavior before vehicle motion has rendered that behavior’s previous set of votes obsolete or erroneous, and that votes are received often enough from all behaviors that synchronization of their votes is not a concern.
Section 3.3: DAMN Command Arbitration Methods

Figure 3-12: Orientation selection process in DAMN.
  a&b) votes from two behaviors are sent to the arbiter,
  c) the arbiter combines the votes as a weighted sum,
  d) the votes are smoothed an interpolated maximum value is selected.
3.3.3 Effect Arbitration

In the third DAMN arbitration scheme, each behavior votes for or against various alternatives in an abstract command space, i.e., they vote for the desired effect of the mechanism being controlled rather than the direct control of the mechanism’s actuators. The arbiter effectively synchronizes the votes by maintaining a consistent command space in which those votes are initially represented, so that votes are counted correctly even after significant vehicle motion by re-mapping them into the current actuator frame of reference, and obsolete votes are not counted at all. However, once a behavior’s votes have become obsolete, that behavior has no effect on the decision-making process until it issues a new set of votes.

For example, a field of regard arbiter and its associated behaviors have been implemented and used for the control of a pair of stereo cameras on a pan/tilt platform. Behaviors vote for different possible field of regard polygons, which are camera fields of view mapped on to the ground plane resulting in a trapezoidal shape, as shown in Figure 3-13. The arbiter maintains a local map of these votes and transforms them as the vehicle moves. At each iteration, these votes are mapped into a pan-tilt space and arbitration takes place within the actuator space. As the vehicle moves, the arbiter updates its commands so that the camera continues to point at the desired location, thus accounting for vehicle movement. The darkest polygon in the figure corresponds to the field of regard formed by the pan and tilt angles selected by the arbiter.

![Figure 3-13: Voting for field of regard polygons, mapped into pan-tilt space for arbitration. Darker polygons have higher vote values; striped polygons represent negative values.](image)

3.4 Limitations of Command Arbitration

3.4.1 Vote Semantics

The DAMN command fusion arbitration scheme has proven to be very effective in facilitating the integration of a wide variety of different vehicle navigation subsystems; it derives much of its power and flexibility from the voting interface between behaviors and their arbiters. On the one hand, the voting interface should be simple enough so that virtually any module should be able to conform to it, regardless of the algorithms and representations used internally within that module;
on the other hand, the interface must be complex enough so that an arbiter is capable of making rational and coherent decisions based upon the information provided to it.

In a centralized planner, the semantics of the decision-making process are often clear, such as maximizing an objective function. Traded control systems such as prioritized behavior-based systems are also relatively understandable since only one behavior is making the decision at any one time, although the emergent properties of such a system are not well-defined. The interactions between elements of other shared control systems such as potential fields are not well-behaved in cluttered environments.

The form of shared control used in DAMN is more structured in that the arbitration takes the form of finding a maximum rather than a mean. However, the semantics of the vote values, which ranged from +1 to -1, are not well defined. One behavior might mean a value of 0 to indicate an undesirable action and a vote of -1 to mean do not take that action under any circumstances, while another behavior might use a vote of 0 to express indifference and -1 to merely mean that the action is undesirable. As DAMN is used for increasingly complex tasks where many behaviors may be issuing votes concurrently, there is a greater need to have the semantics of the voting and arbitration process carefully defined in order to provide a system whose effects are well-defined and well-behaved.

**3.4.2 Control Issues**

When dealing with a physical system such as a mobile robot, it is also important to consider aspects of control such as stability and the limitations and constraints of the physical plant, such as the non-holonomic constraints of a wheeled vehicle. Whether choosing a command output in priority-based arbitration, combining votes in the orientation selection methods, or summing vectors in potential fields, the arbiter is selecting among actions proposed by the behaviors which are not physically realizable. This is represented by the state in Figure 3-14 designated as unreachable, due to the fact the operator that would take the vehicle to that state from the current state could not be achieved given the dynamics of the system; for example, it may require a change in the commanded curvature which exceeds the steering wheel actuator’s torque capabilities. Another important difference between the commanded and actual vehicle trajectories is due to the delays inherent in any system, arising from latencies in data acquisition, data processing, intermodule communications, and actuator response. Together with the continuous motion of the vehicle, these delays imply that by the time the command is being executed the vehicle is no longer in the current state but actually in a future state, i.e. different position, heading velocity, turn rate, etc. Reactive systems hope to avoid this problem by reducing latencies, although any latency becomes problematic at a great enough speed. An asynchronous distributed system presents an additional challenge in that, in general, the size of these latencies will be different for each behavior due to varying processing needs and sensor frame rates, so that each behavior will be issuing votes when the vehicle is passing through various intermediate states between the initial and current vehicle states. If the various latencies of the system are not accounted for, the vehicle control will be unstable.
Figure 3-14: Control problems arising from dynamic state space.
Behaviors vote at intermediate states, arbiter processes votes at current state, commands have effect at future state.

The ideal vehicle response when it is commanded to change from the current curvature $\kappa_1$ to a new curvature $\kappa_2$ is shown on the left in Figure a. However, to faithfully execute the commanded change in curvature would require instantaneous computation and communication, as well as an infinite acceleration of the steering wheel actuator. In reality, the executed turn would be as shown on the right in the figure, with the small vehicle indicating its pose after following that path for its assumed duration. The result is that rather than being able to execute the arcs depicted in Figure b, the actual space of executable turns varies with current speed and turn radius to yield a command space as shown in Figure c, which represents the future trajectories possible when the vehicle is currently making an extreme left turn.

Stable control requires that the system anticipate these latencies and allow the behaviors to effectively vote on turns that are kinematically achievable and which originate from the point where the vehicle will actually be when the command is executed, as was initially demonstrated and implemented in the work described in [35]. In order to plan paths that a vehicle will be capable of following accurately, it is necessary to impose the constraint of continuous curvature as a function of $s$, the distance travelled along the path. Two families of curves which obey this constraint have been investigated for the use of mobile navigation: clothoids [31] and cubic spirals [32]. Clothoids, or Cornu spirals, are curves whose curvature varies linearly with path length: $\kappa = k(s) + \kappa_0$, illustrated in Figure 3-15d. The parameter $k$ specifies the sharpness, or rate of change of curvature.
Given an initial vehicle pose \((x_0, y_0, \theta_0)\), the pose at any point along a clothoid of sharpness \(k\) is given by:

\[
\begin{align*}
\Theta(s) &= \frac{1}{2}ks^2 + \kappa_0 s + \theta_0 \\
x(s) &= x_0 + \int_0^s \cos \left( \frac{1}{2}k\sigma^2 + \kappa_0 \sigma + \theta_0 \right) d\sigma \\
y(s) &= y_0 + \int_0^s \sin \left( \frac{1}{2}k\sigma^2 + \kappa_0 \sigma + \theta_0 \right) d\sigma
\end{align*}
\]

Although \(x(s)\) and \(y(s)\) do not have a closed-form solution, they can be calculated at small regular intervals of \(\Delta s\) for the purpose of tracing out a path. This numerical integration need only be computed once and the poses can then be mapped into the current reference frame of the vehicle with a simple matrix transformation.

---

**Figure 3-15**: Effect of system dynamics on vehicle trajectory.

- a) Ideal response vs. actual response to commanded change in curvature,
- b) turn radii under ideal conditions,
- c) turn radii with delay and initial hard left turn radius,
- d) clothoid spiral of constantly changing curvature
Behavior-based systems do not account for vehicle dynamics and non-holonomic constraints, producing commands that may not be even approximately followed by the system being controlled. DAMN behaviors communicating with a command arbiter, however, generate votes within an actuator command space that reflects the types of commands which the arbiter will actually be issuing; for example turn behaviors vote for arcs of constant curvature rather than the linear trajectories produced by systems such as Subsumption Architecture, Motor Schemas, or the Vector Field Histogram. Thus, DAMN behaviors must have knowledge of the command space in order to be to evaluate candidate actions, as shown in Figure 3-16. The advantage is that at least some non-holonomic constraints of the system are considered in the decision-making process; the disadvantage is that each behavior must be modified if the system is to be used for a vehicle with different kinematic and dynamic constraints.

![Figure 3-16](image)

If behaviors were to be able to account for all of the system dynamics, they would require a great amount of state information, both from the vehicle and from the arbiter. In addition, each behavior would need to have a detailed model of the dynamics and kinematics of the system and apply it to the state information, representing a considerable duplication of effort. Such a system would also be difficult to develop and maintain. Instead, the arbiter can take all of this information into account within a single interchangeable module; the behaviors simply must then provide enough external domain knowledge to allow the arbiter to perform adequate reasoning about the internal system state and dynamics. In particular, the state space within which the arbiter reasons must be sufficient to allow the exploration of the entire actuator state space.

**3.4.3 Synchronization**

Synchronization allows reasoning to be coordinated and therefore coherent, but reduces the throughput of the system as modules must wait for a signal from the system clock in order to remain synchronized. Allowing the modules in a distributed architecture to operate asynchronously, each at the greatest rate of which they are capable, maximizes throughput and therefore reactivity. Some architectures that use priority-based arbitration, such as the Subsumption Architecture, have no coordination between behaviors and therefore do not require
synchronization; however, they are unable to produce coherent output for complex tasks with multiple objectives.

As we have seen, command fusion allows for the simultaneous satisfaction of more than one goal. However, if there is no synchronization between behaviors, then their votes are produced based on different system states. For example, consider the case in Figure 3-17a, where BEHAVIOR 1 sends votes to the turn arbiter from one location, and then BEHAVIOR 2 sends votes a moment later when the vehicle is in a different location. Because of the translation and rotation of the vehicle, the arcs being voted upon by the two behaviors are actually different, so that the semantics of combining the two sets of votes is ill-defined and may yield unpredictable results. This problem is exacerbated in systems such as the camera field of regard arbiter shown in Figure 3-17b, where the physical system is capable of greater rotational velocities.

![Figure 3-17](image)

**Figure 3-17**: Unsynchronized behavior voting.

a) turn votes b) field of regard votes.

### 3.4.4 Representation of Uncertainty

Domains such as mobile robot navigation necessarily contain a great deal of uncertainty. One source of uncertainty is in the sensing of the environment, as illustrated by the uncertainty in the location of an object in Figure 3-18a. Another source of uncertainty is in the sensing of internal state; for example increasing uncertainty in a vehicle’s position as it moves, indicated by the growing ellipses in Figure 3-18b. A third source of uncertainty is in our knowledge of the effects of our actions because of an imperfect or incomplete model of the system; for example, wheel slippage may yield uncertain vehicle control, as shown in Figure 3-18c.

![Figure 3-18](image)

**Figure 3-18**: Sources of uncertainty.

a) external sensing, b) internal position, and c) control.
A reactive architecture does not represent or explicitly deal with uncertainty but instead just samples the world at a high rate so that the most current information is always used. This is the approach taken in the DAMN actuation arbitration scheme, with the assumption that a behavior would update its votes before the information in the previous votes became obsolete. If this assumption is valid, then the system is sufficient for the purpose and due to its simplicity is more robust than a system that uses complex internal representations; however, if the assumption is false and behaviors do not operate at a rate commensurate with vehicle speed, then uncertainties and inaccuracies will have a significant effect before the next update is received. These uncertainties are accounted for in an ad hoc manner in fuzzy logic systems, including the DAMN turn and pan/tilt arbiters, by “fuzzifying” the inputs to the system and using fuzzy reasoning to determine an appropriate output [30], as was shown in Figure 3-7. Similarly, potential fields implicitly deal with uncertainty by virtue of the field’s extension from a point.

3.5 Utility Fusion

A new means of action selection is introduced here as an alternative to both the sensor fusion and command fusion approaches. Evidence concerning the desirability of possible world states is obtained from multiple independent sources and combined via utility fusion. The determination of what the next action taken should be is then based on this combined evidence. In this paradigm, behaviors do not select or express preferences for actions but instead determine the utility of possible world states. It is then the responsibility of the arbiter to determine which states are actually attainable and how to go about achieving them.

Utility fusion does not to create a world model as sensor fusion systems do. For example, certainty grids [48] combines sensory input to create an occupancy grid containing the probability that each cell contains an obstacle. The information combined and stored by the utility fusion arbiter does not represent to a sensed feature of the world, but rather the desirability of being in a particular state according to some criterion defined by the behavior. The processing of sensory data is still distributed among behaviors, so the presence of bottlenecks and other difficulties associated with sensor fusion are avoided.

For example, a utility map-based path arbiter for steering control has been developed. Behaviors communicating with the path arbiter vote on the desirability of various possible vehicle locations, and the arbiter maintains a local map of these votes. Figure 3-19 shows polygons of positive utility that a road-following behavior has sent based on detected road location, with the greatest value being the polygon closest to the center of the detected road, and polygons of negative utility that an obstacle avoidance behavior has sent, with the greatest value being the polygon closest to the center of the detected obstacle. Based on the vehicle’s current state, the path arbiter evaluates the possible trajectories which may be followed, shown in the figure as arcs emanating from the vehicle. The utilities are summed along each arc, and the arbiter selects that one for which the total is the greatest.
Utility theory gives a unified conceptual framework for defining votes and weights and dealing with uncertainty. Because we are attempting to decide which among a set of possible actions to take, it is natural to make judgments on the usefulness of each action based on its consequences. If we assign a utility measure $U(c)$ for each possible consequence of an action, then the expected utility for an action $a$ is:

$$U(a) = \sum_{c} U(c) \cdot P(c|a, e)$$

where $P(c|a, e)$ is the probability that consequence $c$ will occur, given that we have observed evidence $e$ and taken action $a$ [56]. Thus, if we can define these utilities and probabilities, we can then apply the Maximum Expected Utility (MEU) criterion to select the optimal action based on our current information.

Unlike command arbitration or command fusion systems, the utility fusion arbiter does not simply select among or combine actions proposed by behaviors. Instead, the utility fusion arbiter is provided with much richer evaluation information from behaviors, thus allowing for intelligent decision-making to take place within the arbiter. The arbiter accumulates these evaluations from the behaviors and defers decision-making until the latest possible stage based on the combined evidence, so that the limitations of command fusion systems may be overcome.

### 3.5.1 Vote Semantics

In order to apply utility theory to the problem of evidence combination and action selection for mobile navigation tasks, a means for defining the utility functions must be provided. Because behaviors are defined in order to achieve some task, there must be at least an implicit measure of “goodness” or utility with respect to that task. For example, an obstacle avoidance behavior associates a large negative utility with occupying space determined to contain an obstacle, and a
lesser negative utility with being close to the obstacle, as illustrated in Figure 3-20a. Likewise, proximity to the current goal could be used for the goal-based behaviors, as proximity to the center of a road could be used by road following modules, as in Figure 3-20b. Utility theory imbues behavior votes with a relatively intuitive and natural meaning, thus facilitating the development of the voting scheme for a module, as well as the integration of the votes from these various behaviors.

Figure 3-20 : Utilities measures and uncertainties based on detection of detected features. a) obstacle, b) road. Darker polygons have higher utility; stripes indicate negative utilities

In addition, because action evaluation is now performed by the arbiter, behaviors need not know which actions the system is capable of; a behavior can express the utility of a desired world state independently of which actions would need to be taken to achieve it. As is shown schematically in Figure 3-21, a behavior only contains the domain and procedural knowledge needed for evaluating possible world states in the context of the task for which it is responsible. This provides greater modularity and interchangeability of behaviors; for example, a behavior developed for a vehicle with Ackerman steering could be reused as is for a system to control and omnidirectional robot.

Figure 3-21 : DAMN Arbiter evaluates candidate actions using utility map information. Behaviors do not need any knowledge of the dynamics or kinematics of the control system.
3.5.2 Control Issues

One advantage of map-based utility fusion over command fusion is that the dynamics of the system being controlled can be fully modeled and accounted for by the central reasoning process, providing greater control accuracy and stability. For example, the arbiter can use knowledge of its own processing latencies as well as delays inherent in the control system to compensate for them via predictive control. Because the arbiter is evaluating arcs rather than the behaviors, it can do so from the position at which the vehicle is expected to have when the next command is actually executed, indicated by the lighter vehicle outline in Figure 3-22. The non-holonomic constraints can be modeled and accounted for more accurately by the arbiter, using current and predicted vehicle state to determine clothoids to be evaluated as in Figure 3-22, rather than simple arcs which cannot be followed. With this scheme, plant dynamics may be fully accounted for, and vote obsolescence only becomes an issue if the vehicle is moving faster than information can be collected and processed by the behavior, an unavoidable limitation of any control system.

![Figure 3-22](image-url) : Map-based path arbiter voting.
Darker polygons reflect higher vote utility values; striped polygons indicate negative utilities.
Arcs indicate clothoid trajectories evaluated by the arbiter.

3.5.3 Synchronization

A map-based utility arbiter solves the problem of unsynchronized behaviors because the information received from them is not time dependent. The actuator and effect arbiters received votes for actions expressed in a vehicle-centered frame of reference, so that their semantics were ill-defined unless executed at the time of the vote, but the utility arbiter is receiving votes for external world states whose meaning is well-defined independent of the current vehicle state. The use of a map allows synchronization of the votes to occur within the arbiter without imposing a central clock that the behaviors must follow.

The external location-based scheme used in the path arbiter is capable of maintaining a consistent interpretation of the votes received and correctly coordinating votes received at different times and from different locations, updating them as the vehicle moves. This process may be repeated as the vehicle moves to perform action selection based on the most recent information available,
without any new information from the behaviors being immediately necessary. The map maintained by the arbiter provides some persistence, so that it is not necessary for a behavior to remember previously detected objects of interest and their accompanying utility functions. A behavior may send new utilities to be added to the map along with previously determined utilities, or, if desired, a behavior may issue new utilities which override previous ones in order to account for updated information. For example, a behavior can update the location of a moving object by sending a new utility to replace the old one with the outdated location.

### 3.5.4 Representation of Uncertainty

Utility theory teases apart the value of the consequence of an action from the probability that the consequence will occur and provides a Bayesian framework for reasoning about uncertainty. By explicitly representing and reasoning about uncertainty within the decision-making processes, a system can be created whose effects are well-defined and well-behaved. The term $P(c|a,e)$ in the equation above accounts for the fact that the consequence $c$ of action $a$ is uncertain; probability distributions can be used to represent the various sources of uncertainty and Bayesian evidence combination techniques can then be used to combine those probabilities [10].

By casting the voting scheme for this class of arbiter within the framework of utility theory, uncertainty within the system is explicitly represented and reasoned about within the decision-making processes. Each behavior votes for the subjective utility of the vehicle being in the various particular locations of concern to that behavior, e.g. obstacle locations or road locations. The behavior may also express any uncertainty associated with the perception process, using multiple polygons of increasing uncertainty as in Figure 3-20, rather than single points specifying exact locations. The arbiter can then use utility theory to reason explicitly about the uncertainty in position and control of the vehicle and apply the Maximum Expected Utility criterion to select the optimal action based on current information.

### 3.5.5 Limitations of Utility Fusion

Utility fusion represents a compromise between sensor fusion and command fusion, in order to retain some of the properties of coherence and rationality of the former while achieving the responsiveness and robustness of the latter. However, as with any compromise, trade-offs are made which result in being neither as coherent as a purely centralized system nor as responsive as a purely distributed system. The central arbiter is performing more complex reasoning than a command fusion arbiter, giving the approach many of its strengths but also introducing a potential bottleneck, although one not as severe as that represented by sensor fusion. The constant re-evaluation of the situation to decide upon a new action makes the system more reactive and robust than one which follows a pre-planned path, but the path followed will in general be less optimal and this piecewise planner may not be complete, potentially making the system unable to achieve its goal.
In addition, the utility arbiter does not deal with uncertainty in an entirely satisfactory manner. As a behavior informs the arbiter of new utilities, they either replace old ones from that behavior or are included in the map together without any sort of combination taking place. When state space trajectories are evaluated, their expected utilities are summed. While this is correct for independent utilities corresponding to distinct objects, uncertainties for the same object should be combined via Bayes’ theorem [10]. However, this is essentially what takes place in sensor fusion systems and was avoided due to throughput requirements of the system. Instead, behaviors are responsible for combining sensory information as appropriate.

While it was hoped that utility theory would provide an objective means of combining information from disparate sources, utilities are still defined in a subjective manner, according to intuition and experience, just as weights are defined in a command fusion system. This appears to be a fundamental problem of multiple-objective systems, whether distributed or centralized, which is not solved by utility theory. Although determining the relative values of utilities from different behaviors remains more art than science, the definition of utility values within a behavior can be made using intuitive measures such as distance, as described above in Section 3.5.1. For example, an obstacle avoidance behavior detects an obstacle, it notifies the arbiter of a very large negative utility with some uncertainty due to sensing, as well as a utility with a smaller negative value to indicate the undesirability of being too close to an obstacle for reasons such as occlusion of unsensed terrain. Additionally, there is no need to normalize values; thus, some utilities can take on very large, effectively infinite values (e.g., for an obstacle) without effecting the dynamic range of the remaining utilities.

Other limitations of the system are not intrinsic to utility fusion per se, but rather are limitations of the system as implemented, in particular the choice of using a two-dimensional map to represent utilities that only have x and y values associated with them. Other dimensions such as orientation could also be added to the representation, yielding a planner capable of planning in configuration space. However, any added dimensionality could significantly impact the responsiveness of the system; it is for this reason that behaviors, which do not pose a threat of being a central bottleneck, are left with the responsibility to update utilities in a timely fashion as the world and vehicle state changes.

Time is another dimension not explicitly represented by the map-based utility arbiter. Although the arbiter is able to explicitly reason about the movement of the vehicle, the utilities in its map are static so that it is necessary for behaviors to quickly update the location of a utility, even when the object to which it corresponds is moving with a known trajectory. In order to perform a limited form of time-dependent planning, the representation for utilities could be extended to include velocities when known, and the arbiter would then update the position of these utilities as time progresses, as well as in anticipation of system latencies as was done for predictive control. When evaluating a trajectory, the arbiter would also have to account for the time taken for the vehicle to move along the path, updating the predicted position of utilities as the position being evaluated progresses.
3.6 Conclusion

Because reactivity is essential for any real-time system, we must shun the sensing and planning bottlenecks of centralized systems. If we are to avoid sensor fusion, the system must combine command inputs to determine an appropriate course of action. However, priority-based arbitration only allows one module to affect control at any given time. Priority-based arbitration schemes such as that used in the Subsumption Architecture [13] have been successful in achieving “insect-level intelligence.” Their simplicity allows the resulting system to be completely distributed and therefore contain no bottlenecks, thus maximizing responsiveness to the environment. However, it is not clear this scheme can be extended to obtain higher-level functionality in more complex tasks and domains. One of the requirements for a robot control system is that it be capable of satisfying multiple, possibly conflicting goals [67]. While having one behavior’s commands completely override another’s may be an effective scheme for choosing among incompatible commands, it does not provide an adequate means for dealing with multiple goals that can and should be satisfied simultaneously.

Command fusion provides a mechanism for the concurrent satisfaction of multiple goals, and allows modules to be completely independent, thus allowing incremental, evolutionary system development. In systems that perform command fusion, the decision-making process is based on combining command inputs from multiple behaviors, and is therefore able to simultaneously satisfy multiple goals. Such systems are largely distributed, but they must also contain a central module that receives commands from the various behaviors and combines them in some manner. Motor schemas provide a general framework for command fusion, but because they are implemented as potential fields they are subject to the problems of command averaging and local minima, among others.

As an alternative to potential fields methods, DAMN arbiters combine votes for and against each of a set of candidate commands, either directly in actuator space or indirectly in effect space. This method has the benefit of being very simple and straightforward, so that many different types of modules may easily be incorporated within the architecture. However, this simplicity comes at the cost of some assumptions that must hold if the system is to operate correctly; vehicle dynamics and system latencies must be negligible, and behaviors must be capable of updating their votes before the previous votes become incorrect and before synchronization becomes a problem. Uncertainty is dealt with in an ad hoc manner using techniques similar to those used in fuzzy logic systems.

A new means of action selection via utility fusion is introduced as a solution to the shortcomings of behavior-based systems. Instead of voting for actions, behaviors indicate the utility of various possible world states. The arbiter combines these utilities and determines the next action based on the maximization of expected utility, thus providing a unified conceptual framework for defining the semantics of votes and for dealing with uncertainty. By explicitly representing and reasoning about uncertainty within the decision-making processes, a system can be created whose effects are well-defined and well-behaved.
The utility arbiter can use models of the system being controlled to determine which states are actually attainable, and to increase the accuracy and stability of control. In particular, the map-based utility arbiter gathers information from behaviors about the desirability of possible vehicle locations and then evaluates candidate trajectories to determine appropriate actions. The arbiter can then use kinematic models of the robot to determine which actions can be commanded without violating non-holonomic constraints, and use and dynamic models of the system to provide greater stability. Behaviors can function without knowledge of the system dynamics, thus increasing their reusability for other systems and types of vehicles.

It would be possible for each behavior in an actuation arbitration scheme to reason about the kinematic and dynamic constraints of the system, and to perform predictive control, at the cost of greatly increasing the complexity and reducing the reusability of the behaviors. In addition, the behaviors would have to receive state information from the arbiter concerning the commands that have been sent to the controller but not yet executed, and the handshaking involving would further increase complexity. Perhaps the greatest limitation, however, is that behaviors would be forced to operate at a sufficiently high rate so that the prediction and decision-making process can be updated often enough to maintain stable control. Thus behaviors liked the map-based goal planners would be forced to separate into multiple independent processes, effectively duplicating the effort that currently takes place in the utility arbiter.

The utility arbitration system does not perform sensor fusion, but instead the behaviors process data according to their tasks in a distributed fashion. However, the arbiter is more complex than those performing command fusion and resulting the system is thus more centralized than command fusion systems. As a result, systems implemented with utility fusion are less reactive but are better able to anticipate circumstances and to generate coherent action. In DAMN, the processing of sensory information is still distributed among the behaviors, and the utility fusion performed by the arbiter is much simpler and represents much less of a bottleneck than does sensor fusion. By centralizing the action evaluation process, the type of DAMN arbiter is able to address issues discussed such as synchronization of commands and stability of control. Utility theory also provides a semantic framework to define the meanings of votes, and to separate out the effects of uncertainty and reason about them explicitly.
CHAPTER 4: DAMN IMPLEMENTATION

Various methods of command arbitration have been explored within the DAMN framework, using different types of arbiters and vote structures. Mainly due to real-time concerns, sensor fusion is avoided as an integral part of the system, although, when appropriate, it may be used within behaviors to overcome ambiguities and inconsistencies in data. However, previous architectures that perform command arbitration make suboptimal decisions and, as discussed in Chapter 3, are particularly problematic when multiple factors must be simultaneously considered.

Unlike behavior-based systems that use priorities to effect a traded control system, DAMN takes a shared control approach where several modules concurrently have some responsibility for control of the robot. Input from all behaviors are used in determining what the next action should be, so that there are multiple influences on the decision-making process, and compromises are made to satisfy as many objectives as possible. There are currently four different arbitration schemes in use in DAMN, each with their own advantages and disadvantages. Constraint arbitration is the simplest and most distributed; it is similar to dynamic priority-based arbitration and has been used for speed control. Actuation and effect arbitration are forms of command fusion that have been used to implement a turn arbiter and field of regard arbiter, respectively. Finally, a map-based utility arbiter is introduced as a new means of action selection via evidence fusion.

Within the framework of DAMN, behaviors must be defined to provide the task-specific knowledge for the domain. Each behavior runs completely independently and asynchronously, using whichever sensor data and processing algorithms are best suited to the task, and providing votes to the arbiter each at its own rate and according to its own time constraints. While the behaviors are integrating within the DAMN framework, they are not an intrinsic part of the architecture. Their implementation is described to illustrate how various forms of reasoning and combined within DAMN.

4.1 Arbiters

In a distributed architecture, it is necessary to decide which behaviors should be controlling the vehicle at any given time. In some architectures, this is achieved by having priorities assigned to each behavior; of all the behaviors issuing commands, the one with the highest priority is in control and the rest are ignored [13] [60]. In order to allow multiple considerations to affect vehicle actions concurrently, DAMN instead uses a scheme where each behavior votes for or against each of a set of possible vehicle actions. An arbiter then performs command fusion to select the most appropriate action. While all votes must pass through the command arbiter before an action is taken, the function provided by the arbiter is fairly simple and does not represent the centralized bottleneck of more traditional systems.

There is usually a one-to-one mapping between an arbiter and an actuator to be controlled, such as steering or speed, but an arbiter may control multiple actuators when they are interdependent and must be jointly controlled, as is the case with pan/tilt camera control. Each arbiter operates...
separately from all other arbiters, although they may communicate their outputs to one another for loose coordination as with speed and steering control. Likewise, behaviors are usually concerned only with the control of one particular actuator and sends its votes to the arbiter responsible for the control of that actuator, but a behavior may also send votes to more than one arbiter if appropriate, and for convenience an arbiter may also send multiple independent sets of votes to a single arbiter. A non-voting module may also communicate with an arbiter in order to receive informations such as the output commands and the feedforward predictions of internal state calculated by the arbiter.

For the sake of efficiency and simplicity, the command spaces in DAMN are partitioned into discrete action choices, thus allowing the approximate representation of any arbitrary voting function. In the case of the turn arbiter, the behaviors vote directly for discretized actuator commands; for the field of regard arbiter, behaviors vote for discrete choices of camera viewing polygons relative to the vehicle position, and for the path arbiter the behaviors vote for the utility of passing through absolute vehicle positions. Each of these represent increasing abstraction from the direct control of actuators, which is more appropriate for all behaviors except those operating at the very lowest level of control, which can and should be incorporated directly into the arbiter itself. This command abstraction also has the important benefit that a voting behavior need not be concerned with the kinematics and dynamics of the vehicle, so that the same obstacle avoidance behavior used for the control of a vehicle with Ackerman steering may directly be ported for use on an omnidirectional robot.

Although the internal command representations used by the arbiters are discrete, the inputs to them from the voting behaviors need not be, and indeed the interfaces allow for the specification of continuous voting functions. This introduces the questions of whether and how accurately these inputs need to be reconstructed for the sake of precise and stable control, and how to best effect it. A related question that arises in both continuous and discrete systems is whether and how smoothing should be performed.

### 4.1.1 Turn Arbiter

The turn arbiter is an instance of the actuation arbitration scheme whose command space consists of discrete curvatures. The curvature space is discretized into $N$ arcs linearly spaced between zero and the maximum curvature to the right, one arc to steer the vehicle straight ahead, and another $N$ arcs linearly spaced between zero and the maximum curvature to the left, as shown in Figure 4-1.

Initially, the turn command options in DAMN were represented as turn radii ranging from the smallest possible vehicle steer radius for making extreme hard right or left turns to an infinite turn radius for proceeding straight ahead. Since a command space based on turn radius is non-linear and contains a discontinuity for a straight line, the turn command options are instead represented by the multiplicative inverse of turn radius, i.e., curvature, which does not suffer from these problems.
The curvature $\kappa_i$ of each arc of index $i$ is computed as: $\kappa_i = \left( \kappa_{max}/N \right) \cdot (i-N)$, $i = 1, \ldots, N$. Negative curvature values represent left turns. The tightest possible turn that the Navlab vehicles can achieve is a turn radius of 8 meters, so that the value of $\kappa_{max}$ is 0.125 meters$^{-1}$. Providing 25 discrete turn choices, for an incremental value of 0.005 meters$^{-1}$ per arc, was found to be empirically satisfactory; multiplying by 2 to account for both left and right turns and adding 1 to account for straight ahead yields a value of 51 for $N$.

![Figure 4-1: Curvature-based turn command space.](image)

Each turn behavior generates a vote between -1 and +1 for every possible steering command, with negative votes being against and positive votes for a particular command option. The votes generated by each behavior are only recommendations to the arbiter. The arbiter collects the new votes from each behavior that has sent them, and arbitrates these values to determine the command with the highest total vote value. In order to avoid problems with discretization such as biasing and “bang-bang” control (i.e., alternating between discrete values in order to achieve an intermediate value), the arbiter performs sub-pixel interpolation. The details of the process of combining votes and determining a command to be sent to the vehicle controller are given by the following procedure and illustrated in Figure 4-2.

**Operation of Arbiter**

0) **Initialize behavior connections**
   Connect to and register with communications module, and with vehicle controller. Optionally reset vehicle position. Optionally connect to a mode manager. When a new behavior connects to the arbiter, register it as a voting module and send it a identification number to be used for tagging votes with the module ID. If connected to a mode manager, notify it of new behavior.

1) **Collect votes issued by behaviors (Figure 4-2a&b)**
   At regular intervals, collect the votes which have been asynchronously sent by the various behaviors. Only the most recently sent set of votes is used.
a) Behavior 1, weight = 0.8
   peak curvature = 0.04

b) Behavior 2, weight = 0.2
   peak curvature = 0.0

c) Weighted Sum
   peak curvature = 0.035

d) Smoothed & Interpolated
   peak curvature=0.033

e) Hypothetical flat region
   mean curvature=0.035

Figure 4-2: Turn arbiter command fusion process.
Histograms of vote value $\nu$ as a function of curvature $\kappa$; arrow indicates
2) Compute weighted sum of votes (Figure 4-2c)

\[
\forall i \in \{1, n\}, V_S[i] = \sum_{b} v_b[i] \cdot w_b \cdot e^{(t_b - \tau_b)/\tau_b}
\]  
Equation 4-1

For each candidate turn command, compute \(V_S[i]\), the sum of the votes \(v_b[i]\) received for that turn by each behavior \(b\), multiplied by that behavior’s weight \(w_b\). Each behavior also has an associated “half-life” \(\tau_b\); if the time \(t_b\) since the votes from that behavior were last received exceeds this half-life, then the votes are subject to an exponential decay.

3) Convolve summed votes with a Gaussian mask (Figure 4-2d)

\[
\forall i, j \in \{1, n\}, V_G[i] = \sum_{j} V_S[j] \cdot \left(e^{-((\kappa_j - \kappa)^2/2\sigma^2)}/2\Pi\sigma\right)
\]  
Equation 4-2

For each weighted summed vote \(V_S[i]\) and associated turn command of curvature \(\kappa_i\), convolve a normal distribution in curvature space with the votes for that curvature and neighboring curvatures and store the result as \(V_G[i]\). This smoothing allows for the subpixel interpolation performed in Step 5 (ii) and also has the effect of favoring consecutive sequences of turn commands with positive votes over a single highly positive vote surrounded by turn commands with significantly lower votes, thus accounting for uncertainty in sensing and control in an ad hoc manner. The value of \(\sigma\) used in vehicle experiments was 0.005.

4) Find maximum vote value (Figure 4-2d)

\[V_M = V_G[\Lambda] | \forall i(V_G[\Lambda] \geq V_G[i])\]  
Equation 4-3

Set the maximum vote value, \(V_M\), to be the value of \(V_G[\Lambda]\) such that no other \(V_G[i]\) has a greater value. This is indicated by the arrow in Figure 4-2d.

5) Interpolate commanded curvature (Figure 4-2d&c)

i) If at least one command neighboring \(\kappa_\Lambda\) has a vote value within \(\epsilon\) of \(V_M\), i.e. \((V_{\Lambda-1} \equiv V_\Lambda) \lor (V_{\Lambda+1} \equiv V_\Lambda)\), then find the sequence of consecutive turn commands such that their vote values are within \(\epsilon\) of \(V_M\), let their indices range from \(\Lambda - \alpha\) to \(\Lambda + \beta\), with \(\alpha, \beta \geq 0\); define \(\mu\) to be the mean of these indices: \(\mu = ((\Lambda - \alpha) + (\Lambda + \beta)) / 2\). Then set the turn command to be the corresponding curvature \(\kappa_\mu\). To illustrate this, a “flat” region where the votes are of the same value was created in Figure 4-2e; the curvature corresponding to the mean of that region is selected as the output turn command.

ii) Otherwise, the maximum is not in a flat region, so it assumed that the distribution of votes, which has been convolved with a gaussian in Step 3, can
locally be approximated by a parabola defined by three points: the maximum vote value for the chosen curvature and the vote values for the two adjacent curvatures. The curvature corresponding to the peak of this parabola is then chosen as the commanded turn, as shown in Figure 4-2d. Note that this is shifted somewhat toward the right from the arrow indicating the maximum discrete value.

### 4.1.2 Speed Arbiter

Two simple schemes for speed arbitration have been implemented, although there has been insufficient opportunity to adequately compare the results on the vehicle testbed.

*Speed arbitration as a function of turn choice confidence*

Initially the commanded speed was determined as a function of the votes for the chosen turn radius. A user-specified maximum vehicle speed was multiplied by aggregate confidence of the behaviors in the chosen turn radius, as reflected by the normalized weighted sum of the votes for that turn radius. The resulting value was the speed command issued, as shown in Figure 4-3. This method has the drawback that the importance of a turn vote cannot be established independently of the vehicle speed. For example, a behavior may determine that it is very important to make a hard left turn while proceeding slowly; this would be impossible to specify in this combined arbitration scheme. The issues involved are discussed further in Section 4.1.5, “Coordination of Arbiters,” on page 81.

![Figure 4-3: Commanded speed proportional to maximum turn vote.](image)

Circle size indicates vote magnitude from 0 to 1; striped circles represent negative values.

*Speed arbitration as constraint satisfaction*

An entirely separate DAMN constraint space speed arbiter has been developed which has its own set of associated behaviors that vote for the maximum allowable speed which satisfies the various constraints that are to be considered. A simple continuous space is used; each behavior indicates the highest allowable speed that satisfies its limits, and the arbiter selects the minimum of these so that all speed constraints are satisfied, as illustrated in Figure 4-4. Thus, the turn behaviors can vote for turn commands without concern that the absolute magnitude of their votes will affect vehicle speed.
4.1.3 Field of Regard Arbiter

A field of regard arbiter and its associated behaviors have also been implemented and used for the control of a pair of stereo cameras on a pan/tilt platform. This is an example of effect arbitration, because instead of voting directly for the desired pan and tilt angles of the camera platform, behaviors vote in a field of regard command space, which is an abstraction of the pan/tilt command space. Field of regard polygons are camera fields of view mapped on to the ground plane, resulting in the trapezoidal shape seen in Figure 4-5.

If an actuation arbitration scheme were used, it would be difficult to ensure rational coherent behavior using such a pan/tilt command representation because of the sparseness of the information provided to the arbiter and its ephemeral quality due to vehicle motion. In particular, the desired pan and tilt angles change very quickly as the vehicle moves and it would not be possible for many of the voting to issue votes at a satisfactory rate. For example, consider the situation shown in Figure 4-5 where a behavior has voted to look to the left relative to the position of the vehicle at that time, $P_1$, but since then the vehicle has turned to the left and is at position $P_2$, so that looking to the left as the dashed arrow indicates no longer realizes the behavior’s intentions and it would actually be better to look straight from that position, as illustrated by the solid arrow.

Instead, the arbiter receives votes in an effect command space that is an abstraction of the actuator space. Behaviors vote for different possible field of regard polygons, indicated in Figure 4-6. The arbiter maintains a local map of these field of regard votes and transforms them as the vehicle moves, thus alleviating the problems caused by allowing behaviors to operate asynchronously. These votes are then mapped into the pan and tilt angles suggested by the grid. Because the votes are transformed into new coordinate frames as the vehicle moves, the grid is divided into
subpixels to provide finer granularity in the geometric mapping without incurring additional complexity in the subsequent stages of the voting process. A two dimensional representation is used since pan and tilt are intimately related and cannot possibly be determined independently.

Thus, the behaviors are voting for a desired field of regard, which is an abstraction of the pan and tilt angles that must be commanded to the pan/tilt controller. Therefore, the votes must ultimately be mapped from one domain into the other. This could be done as a final stage after a field of regard polygon has been selected by the arbiter, after the functions of vote combination, smoothing and interpolation have been performed on the ground plane. However, because the mapping from a perspective view to camera angles is non-linear, we have chosen instead to map all votes into the pan/tilt space and perform the arbitration within that space so that the ultimate result has been smoothed and interpolated in the actual command space.

Figure 4-7 illustrates the process of collecting votes for field of regard polygons, mapping them into pan/tilt actuator space, and combining the mapped votes to produce an arbitrated result. In the vehicle experiments, camera tilt was determined based on the lookahead needed to maintain a desired vehicle speed and was held constant at 0.15 radians. Five field of regard polygons were used with pans of 0.0, ±0.25 and ±0.5 radians.

**Operation of Arbiter**

1. **Initialize behavior connections**
   
   Connect to and register with communications module, and with vehicle controller. Optionally reset vehicle position. Optionally connect to a mode manager. When a new behavior connects to the arbiter, register it as a voting module and send it a identification number to be used for tagging votes with the module ID. If connected to a mode manager, notify it of new behavior. The arbiter also sends out the following information to the behaviors so that the definition of the voting space can be specified at start-up rather than compile time:

   - number of Field of Regard polygons to be used in voting
   - locations of the vertices in ground plane for each Field of Regard polygon
   - corresponding pan and tilt for each Field of Regard polygon
1) Collect field of regard votes issued by behaviors (Figure 4-7a)
As with the turn arbiter, each behavior generates a vote between -1 and +1 for each possible command, with negative votes being against and positive votes for a particular command option. At regular intervals, the arbiter collect those new votes which may have been sent asynchronously by the various behaviors for field of regard viewing polygons; only the most recently sent set of votes is used. A set of votes is tagged by the sending behavior with the position of the vehicle from which the votes were generated.

2) Map field of regard votes to pan/tilt command space in current vehicle coordinate frame (Figure 4-7b & c)
For those votes issued relative to a previous vehicle position, transform them into the coordinate frame whose origin is fixed at the current vehicle position. To illustrate, consider the example in Figure 4-7c. When the vehicle was at position P1, the candidate field of regard commands for which votes would have been received are those indicated as F1 in the figure and would have been mapped into the pan tilt space as the votes indicated by V1. If the vehicle is now at position P2, then the candidate field of regard commands are those indicated as F2. If no transformation is done, then the field of regard votes F1 would be directly mapped one for one into F2, but as can be seen in the figure, they do not correspond geometrically.

Figure 4-7: Field of regard voting and arbitration.

a) Votes for field of regard polygons on ground plane based on current vehicle pose,
   b) votes are mapped into pan/tilt command space for arbitration,
   c) votes are tagged with vehicle pose and updated as the vehicle moves.
For each candidate field of regard polygon, compute the pan and tilt angles that would realize them, and map the votes received into the pan/tilt command space. The shape of the field of regard polygon corresponds to the field of view when the camera axis is directed at the center of that polygon. However, because of the nonlinearities in the perspective transformation, this shape is warped as the vehicle moves. We avoid this difficulty by simply mapping a vote for a polygon into the pan and tilt angles that would align the axis with the centroid and ignore the perspective skew. For relatively small angles this has not represented a problem.

3) **Compute weighted sum of votes**

\[
V_S[i] = \sum_{i \in \{1, n\}} v_b[i] \cdot w_b
\]

Equation 4-4

Tilt was being held constant. For each pan command, compute \(V_S[i]\), the sum of the votes \(v_b[i]\) received for that pan by each behavior \(b\), multiplied by that behavior’s weight \(w_b\). Unlike votes to the turn arbiter, these votes are not subject to an exponential decay with time because the geometric transformations ensures the votes’ validity.

4) **Convolve summed votes with a 2D Gaussian mask**

\[
V_G[i] = \sum_i V_S[j] \cdot \left( e^{-\left(\frac{(\phi_j - \phi_i)^2}{2\sigma_\phi^2}\right)} \right) \frac{1}{2\Pi\sigma_\phi}\frac{1}{\sigma_\phi}
\]

Equation 4-5

For each weighted summed vote \(V_S[i]\) and associated pan angle \(\phi_i\), convolve a normal distribution in pan angle space with the votes for that pan and neighboring angles and store the result as \(V_G[i]\). This smoothing allows for the subpixel interpolation performed in Step 6 and also has the effect of favoring consecutive angles with positive votes over a single highly positive vote surrounded by commands with significantly lower votes, thus compensating for uncertainty in sensing and control. The value of \(\sigma\) used in vehicle experiments was 0.20.

5) **Find maximum vote value**

\[
V_M = V_G[\Lambda] | \forall i(V_G[\Lambda] \geq V_G[i])
\]

Equation 4-6

Set the maximum vote value, \(V_M\), to be the value of \(V_G[\Lambda]\) such that no other \(V_G[i]\) has a greater value.
6) **Interpolate commanded pan**

   i) If at least one command neighboring \( \phi_\Lambda \) has a vote value within \( \varepsilon \) of \( V_M \), i.e. \((V_{\Lambda-1} \cong V_\Lambda) \lor (V_{\Lambda+1} \cong V_\Lambda)\), then find the sequence of consecutive pan commands such that their vote values are within \( \varepsilon \) of \( V_M \); let their indices range from \( \Lambda-\alpha \) to \( \Lambda+\beta \), with \( \alpha, \beta \geq 0 \); define \( \mu \) to be the mean of these indices:
   \[
   \mu = \frac{(\Lambda-\alpha) + (\Lambda+\beta)}{2}.
   \]
   Then set the pan command to be the corresponding angle \( \phi_\mu \).

   ii) Otherwise, the maximum is not in a flat region, so it assumed that the distribution of votes, which has been convolved with a gaussian in Step 4, can locally be approximated by a parabola defined by three points: the maximum vote value for the chosen pan angle and the vote values for the two adjacent angles. The angle corresponding to the peak of this parabola is then chosen as the commanded pan.

4.1.4 **Path Arbiter**

To address the problems of synchronization, predictive control, dynamic constraints, etc. discussed in Chapter 3, a map based path arbiter has been implemented as a novel means of voting for and producing steering control. Behaviors communicating with the path arbiter vote on the desirability of various possible vehicle locations, and the arbiter maintains a local map of these votes. The map could either be continuous and composed of points and polygons, or it could be discrete and composed of grid cells. Grids introduce the problem of aliasing, yet they are more efficient and a more natural representation for behaviors that themselves use grids. The solution has been to provide both representations simultaneously, and each behavior uses that one which is more convenient and appropriate, as shown in Figure 4-8.

![Figure 4-8: Continuous and discrete utility map representations.](image)

Darker polygons and grid cells reflect higher vote utility values; stripes indicate negative utilities.

This external location-based scheme is capable of maintaining a consistent interpretation of the votes received and correctly coordinate votes received at different times and from different locations, and updating them as the vehicle state changes, thus eliminating the problems of asynchronous operation described in Section 3.4.3, “Synchronization,” on page 53.
In addition, the map-based space which the arbiter maintains enables it to explore the entire continuous actuator state space, which in turn allows it to evaluate actions which take into account the dynamics of the vehicle and the latencies of the system, thus minimizing the problems described in Section 3.4.2, “Control Issues,” on page 50 that are inherent in any dynamic system. By collecting utilities from behaviors and applying the Maximum Expected Utility criterion to select an optimal actions, the arbiter is able to begin to address the problems associated with ill-defined voting semantics described in Section 3.4.1, “Vote Semantics,” on page 49, as well as provide an explicit representation and reasoning mechanism for dealing with uncertainty, as described in Section 3.4.4, “Representation of Uncertainty,” on page 54.

The vote collection and arbitration process is described in detail below. However, since the representations used by this arbiter are more complex, the creation of the appropriate data structures is described first.

**Initialization of Data Structures**

1) **Initialize Vehicle Trajectories**

   Read in arbiter parameters: total number of arcs $N$, arc resolution $\Delta s$, arc length $L$. Typical values used are $N = 11$, $\Delta s = 1.0$ meters, and $L = 10.0$ meters.

   The curvature $\kappa_i$ of each arc is then defined as:

   $$\kappa_i = C(i) = i \cdot \left(\kappa_{\text{max}} \cdot \frac{(N - 1)}{2}\right) - \kappa_{\text{max}} .$$

   Equation 4-7

   Create $N^2$ arcs, each arc $A_{ij}$ corresponding to commanding a curvature $\kappa_i$ while the vehicle is currently executing an arc of curvature $\kappa_j$, so that the vehicle state and actuator dynamics may be considered when evaluating a possible turn command.

   The vehicle path initially follows a clothoid pattern $\kappa = \tilde{k}(s) + \kappa_0$, where $k$ is the constant rate of change of curvature, $s$ is the distance travelled along the path, and $\kappa_0$ is the initial curvature (see Equation on page 52). Although $x(s)$ and $y(s)$ do not have a closed-form solution, they can be calculated at small regular intervals of $\Delta s$ for the purpose of tracing out a path. This numerical integration need only be computed once and the poses can then be mapped into the current reference frame of the vehicle with a simple matrix transformation.

   For each arc $A_{ij}$ with desired curvature $\kappa_j$ and initial curvature $\kappa_i$, take steps of size $\Delta s$ until the maximum path length $L$ is reached; determine the vehicle pose at each time step $t$ as follows:
Equations 4-8

\[ s_0 = 0, n = 0 \]

\[ s_n = s_{n-1} + \Delta s \]

\[ \text{if } (\bar{\kappa}s_n + \kappa_i) < \kappa_j, \text{ then } \kappa_n = \bar{\kappa}s_n + \kappa_i \]

\[ \text{else } \kappa_n = \kappa_j \text{ and set } \bar{\kappa} = 0 \]

\[ \theta_n = \frac{1}{2} \bar{\kappa}(s_n)^2 + \kappa_0 s_n + \theta_0 \]

\[ x_n = x_{n-1} + \cos(\theta_n) \cdot \Delta s \]

\[ y_n = y_{n-1} + \sin(\theta_n) \cdot \Delta s \]

\[ \text{if } s_n < L \text{ then go to step } i, \text{ else stop } \]

These computations are made assuming that the vehicle is travelling at a rate of 1 meter/second; the actual speed is taken into account during runtime computations as described below in Step 2 (iii) of the “Operation of Arbiter” subsection.

2) Create Vehicle Polygons

Read in arbiter parameters: vehicle safety margin \( m \), vehicle length \( l \), and vehicle width \( w \). Typical values used are \( m = 0.3 \) meters, \( l = 3.9 \) meters, and \( w = 2.5 \) meters. These values are measured from the “guide point” at the center of the rear axle of the vehicle.

\[ d_v = \sqrt{\left(\frac{w}{2} + m\right)^2 + (l + m)^2} \]

\[ \theta_v = \arctan\left(\frac{\frac{w}{2} + m}{l + m}\right) \]

Equation 4-9

\[ \text{ii) For each vehicle pose } (x_n, y_n, \theta_n), \text{ along the arcs created by Equation 4-8 in Step 1, determine the sine and cosine of the angles to the front corners of the vehicle in global coordinates, which is just } \theta_n + \theta_v \text{ for the right corner and } \theta_n - \theta_v \text{ for the left corner; these values are stored for later use so that the trigonometric functions only need to be computed once. The actual positions of the corners are determined later when the vehicle’s speed is known.} \]
3) **Create Drive Straight Bias**

The drive straight bias is a behavior internal to the arbiter that favors steering straight ahead, all else being equal. This is achieved by defining a normal distribution of votes centered at the center arc and decaying with increasing curvature.

0) Read in arbiter parameters: drive straight weight \( w_s \), and drive straight standard deviation \( \sigma_s \). Typical values used are \( w_s = 0.01 \), and \( \sigma_s = 0.05 \). For each arc \( i \), compute the curvature \( \kappa_i \) of each arc according to Equation 4-7.

\[
\kappa_i = \frac{-2}{\sigma_s^2} \cdot \frac{1}{2} \cdot \frac{1}{2} \\
\]

\( i \) Set the drive straight bias for each arc to be:

\[
b_i^s = w_s \cdot e^{-\kappa_i^2/2\sigma_s^2} \quad \text{Equation 4-10}
\]

4) **Initialize Communications**

Connect to and register with communications module, and with vehicle controller. Optionally reset vehicle position. Optionally connect to a mode manager.

**Operation of Arbiter**

0) **Initialize behavior connections**

When a new behavior connects to the arbiter, it identifies itself as either one whose votes will either geometric (i.e., points, lines, and polygons), or grid based in nature. Register the behavior as a voting module and send it an identification number to be used for tagging votes with the module ID. If connected to a mode manager, notify it of new behavior.

If the behavior will be voting using a grid, then create a grid of the size and resolution specified by the behavior; this size must remain fixed. Note that, unlike behaviors registering with the arc-based turn arbiter, these behaviors need no...
information about the internal representations of the arbiter or even of the vehicle command space.

1) **Collect utilities from behaviors (Figure 4-10)**

At regular intervals, the arbiter collects those new utilities which may have been sent asynchronously by the various behaviors. Each utility has a real numbered value, and optionally standard deviations specifying a two-dimensional gaussian uncertainty. A set of utilities is tagged by the sending behavior with the position of the vehicle from which the utilities were generated.

A behavior may simply send new utilities to be added to the map along with previously determined utilities. Or, if desired, a behavior can send new utilities which override previous ones in order to account for updated information. For example, a behavior can update the location of a moving object by sending a new utility to replace the old one with the outdated location. Unless explicitly replaced in this manner, a utility value is maintained and updated in the utility map until the vehicle has passed it by three standard deviations.

The utilities may be represented by geometric constructs (points, lines, and polygons) as in Figure 4-10a&b, in which case they also have an associated two-dimensional gaussian distribution. The utilities may also be represented by a grid as in Figure 4-10c, whose frame of reference is defined such that the cell in the middle column of the bottom row of the array corresponds to the position of the vehicle from which the votes were generated. Both representations are made available so that behaviors may use whichever is more convenient and efficient; the arbiter maintains the two representations independently and later performs utility fusion between them.

![Figure 4-10: Examples of utility votes received from behaviors.](image)

a) negative point utility, b) positive polygonal utility, and c) grid utilities.
2) Perform Evidence Fusion and Generate Votes
   
i) Get current vehicle state from controller, position \((x_t, y_t, \theta_t)\), speed \(v_t\), and curvature \(\kappa_t\). This is depicted by the dark vehicle in Figure 4-11.

   ![Figure 4-11: Integration of utilities along candidate trajectories.](image)

   ii) Predict future vehicle state at time of command execution \((x_{t+1}, y_{t+1}, \theta_{t+1})\), based on an estimated latency \(l\) which is the sum of the specified system lag time and the measured arbiter cycle time. This is depicted by the light vehicle in Figure 4-11.

   \[
   \hat{\theta}_{t+1} = \theta_t - l \cdot v_t \cdot \kappa_t \\
   \hat{\bar{\theta}}_{t+1} = \frac{\theta_t + \hat{\theta}_{t+1}}{2} \\
   \hat{x}_{t+1} = x_t - l \cdot v_t \cdot \sin(\hat{\theta}_{t+1}) \\
   \hat{y}_{t+1} = y_t + l \cdot v_t \cdot \cos(\hat{\bar{\theta}}_{t+1})
   \]

   Equation 4-11

   iii) For each vehicle pose \((x_n, y_n, \theta_n)\) along the arcs created by Equation 4-8 in Initialization Step 1, determine the position of the vehicle guide point in the predicted vehicle reference frame. \(x_n\) and \(y_n\) are multiplied by \(v\) to account for the current vehicle speed. The relative distances and angles computed in Initialization Step 2 are now added to this updated pose point to yield the position of the front corners of the vehicle bounding box:

   \[
   x'_n = x_n \cdot v + d_v \cdot \sin(\theta_n + \theta_v) \\
   y'_n = y_n \cdot v + d_v \cdot \cos(\theta_n + \theta_v)
   \]

   Equation 4-12

   iv) For each behavior with geometry-based utilities, compute the coordinates of each utility in a coordinate reference frame centered on the predicted vehicle position. Let these coordinates be \((x_v', y_v')\). If any utility lies more than three
of its standard deviations behind the vehicle, i.e., \((y_u' - y_n') < 3 \* \sigma_y\), remove it from the behavior’s list of utilities to be considered.

\(v)\) Let arc \(A_{ij}\) be the precomputed clothoid trajectory that transitions from the current vehicle turn curvature \(\kappa_i\) to a final curvature of \(\kappa_j\), for each candidate turn command \(\kappa_j\). For each discrete step along the arc \(A_{ij}\) as computed in Equation 4-8, transform arc point \(n\) to \((x_n', y_n')\) in the current vehicle reference frame. The for each geometric utility \(u\), determine the transformed coordinates \((x_u', y_u')\) of the point that is closest in Mahalanobis distance from \((x_n', y_n')\).

For a point utility this would simply be the point itself, for a line segment it would either be one of the endpoints or where the perpendicular from the arc point intersects the segment, and for a polygon it is the closest point on the perimeter, as in Figure 4-12a.

\[ U(n, u) = \nu \cdot e^{-\frac{((x_u' - x_n')^2 / 2\sigma_x^2) + ((y_u' - y_n')^2 / 2\sigma_y^2))}{2\Pi\sigma_x\sigma_y}} \]  

\textit{Equation 4-13}

The utility values for grid voting behaviors is determined simply by indexing the array with the predicted vehicle position, as shown in Figure 4-12b, and multiplying the result by the utility value \(\nu\); a value of zero is used if the indexed position is outside the bounds of the array.

The total utility for taking action \(a\), i.e., following arc \(A_{ij}\), is then computed by taking the sum of the utilities at each of the points along the arc, multiplied by
a discount factor $\lambda$, $0 < \lambda < 1$, that is used to account for the diminished expected returns of future actions.

$$U(a) = \sum_{u} \sum_{n=1}^{L} \lambda^i \cdot U(n, u)$$  \hspace{1cm} \text{Equation 4-14}

3) Normalize Summed Utilities

The summed utilities for each candidate arc command are normalized by the greatest positive such value, so that no value exceeds $+1$. This is done so that the drive straight and maintain turn biases can be added in at the level of actuator votes and have a constant effect regardless of the magnitude of the utilities. Only the largest positive value is used because it was found that a very large negative value would cause all other values to be nearly equal after normalization, and the biases would then dominate the voting process, which was not the desired effect. If there is a very large positive utility, then it will dominate regardless of the biases.

4) Add Turn Biases [optional]

For each turn command, add in the precomputed drive straight bias $d_s$ as determined by Equation 4-10, and the maintain turn bias $d_m$ which favors continuing along the arc of curvature $\kappa_t$ along which the vehicle is already turning:

$$b_i^m = w_m \cdot e^{-\frac{(\kappa_i^2 - \kappa_t^2)}{2\sigma_s^2}}$$  \hspace{1cm} \text{Equation 4-15}

These biases are optional and their values may be set to 0. Note that the expected utilities have now been mapped into summed votes for turn radii, so that the rest of these steps can proceed exactly as with the turn arbiter, as described in Section 4.1.1, thus effectively applying the Maximum Expected Utility criterion to select the optimal action based on our current information.

5) Find maximum vote value

$$V_M = V_G[\Lambda] \mid \forall i(V_G[\Lambda] \geq V_G[i])$$  \hspace{1cm} \text{Equation 4-16}

Set the maximum vote value, $V_M$, to be the value of $V_G[\Lambda]$ such that no other $V_G[i]$ has a greater value. This is indicated by the arrow in Figure 4-2d.

6) Interpolate commanded curvature

i) If at least one command neighboring $\kappa_\Lambda$ has a vote value within $\epsilon$ of $V_M$, i.e. $(V_{\Lambda-1} \equiv V_\Lambda) \lor (V_{\Lambda+1} \equiv V_\Lambda)$, then find the sequence of consecutive turn commands such that their vote values are within $\epsilon$ of $V_M$, let their indices range from $\Lambda-\alpha$ to $\Lambda+\beta$, with $\alpha, \beta \geq 0$; define $\mu$ to be the mean of these indices:

$$\mu = ((\Lambda-\alpha) + (\Lambda+\beta)) / 2.$$  Then set the turn command to be the corresponding
This is the “flat” region illustrated in Figure 4-2e; the curvature corresponding to the mean of that region is selected as the output turn command.

ii) Otherwise, the maximum is not in a flat region, so it assumed that the distribution of votes, which is a sum of normal distributions, can locally be approximated by a parabola defined by three points: the maximum vote value for the chosen curvature and the vote values for the two adjacent curvatures. The curvature corresponding to the peak of this parabola is then chosen as the commanded turn, as shown in Figure 4-2d. Note that this is shifted somewhat toward the right from the arrow indicating the maximum discrete value.

4.1.5 Coordination of Arbiters

An arbiter typically controls exactly one actuator, such as the steering or speed of a vehicle. However, there exists some interdependence between these control variables, so various means of coordinating action between arbiters have been experimented with.

The loosest coupling of turn and speed arbiters is the scheme where they are entirely separate and do not communicate directly at all. Instead, coordination takes place indirectly through the turn and speed behaviors. Many of the speed behaviors have as one of their inputs the output of the turn arbiter, so that the choice of an appropriate speed is influenced by the currently commanded turn radius. Other speed behaviors instead use the estimated actual turn radius of the vehicle so that they operate in a closed-loop fashion, albeit with greater delays. Likewise, some turn behaviors use the current vehicle speed in deciding upon allowable turn options. This is illustrated in Figure 4-13.

For speed commands, a simple continuous space is used; each behavior simply indicates the highest allowable speed that satisfies its limits, and the arbiter selected the minimum of these so that all speed constraints are satisfied. A two-dimensional arbiter which accepts votes for combined turn and speed commands was considered but rejected as being unwieldy. However, some sort of coordination between turn and speed is highly desirable. Because the choices of turn and speed commands are not completely independent and therefore must be coordinated, many of the speed behaviors have the output of the turn arbiter as one of their inputs, so that the choice of an appropriate speed is influenced by the currently commanded turn radius. Other speed behaviors instead use the estimated actual turn radius of the vehicle so that they operate in a closed-loop fashion, albeit with greater delays. Likewise, some turn behaviors use the current vehicle speed in deciding upon allowable turn options.
Another form of coordination is illustrated in Figure 4-14, where the turn and speed arbiters are more directly coupled by each one receiving as input the output of the other, as well as the state variable being controlled by the other arbiter. However, since domain knowledge is contained within the behaviors, an arbiter’s capability to make use of such information is very limited; for this reason this coupling scheme has not been implemented.

Figure 4-13: Coordination of turn and speed commands via behavior inputs.

Figure 4-14: Coordination of turn and speed arbiters via exchanged state information.

Tighter coupling requires that the speed and turn commands be determined by a single consolidated arbiter, as shown schematically in Figure 4-15. One such arbiter computed a turn command based on arc votes as described in Section 4.1.1. The commanded speed could then
optionally be computed by multiplying the normalized weighted sum for the chosen turn radius by a user-specified maximum vehicle speed, as shown in Figure 4-3 in Section 4.1.2. This is similar to the method used in Motor Schemas where the magnitude of the resultant vector determines the speed of the vehicle. Both methods suffer from the fact that a behavior cannot vote to specify that a given direction is good without also making the vehicle go faster; in general these considerations should be made independently.

![Figure 4-15: Coordination of turn and speed arbiters via consolidation.](image)

Coordination of multiple actuators may also be accomplished by creating a multi-dimensional command space and requiring that behaviors vote for all combinations of their cross-product, as shown in Figure 4-16. However, the combinatorics require prohibitively expensive computations on the part of all behaviors, as well as the arbiter; this is exacerbated as the dimensionality or the resolution within any single dimension increases.

![Figure 4-16: Two-dimensional coordinated speed and turn arbitration.](image)

This problem of combinatorial explosion may be overcome by creating a projection of the multi-dimensional control space into a one-dimensional voting space. This is the approach taken by the field of regard arbiter, which accepts votes for discrete viewing polygons and maps that into a two-dimensional pan/tilt space, as described in Section 4.1.3; thus, multiple interdependent actuators may be controlled jointly in an efficient manner.

The map-based utility arbitration scheme provides yet another possible means of coordinating multiple degrees of freedom. As described Step 2 (iii) of the “Operation of Arbiter” subsection of Section 4.1.4, the trajectories evaluated depend upon the current vehicle speed. If instead the arbiter were to evaluate trajectories at several different speeds, it could then choose that turn and speed combination which maximizes expected utility overall. While this induces some combinatorics within the arbiter, it can be partially compensated for by precomputing the
trajectories at the pre-determined speeds. One significant advantage of this scheme, however, is that behaviors incur no greater structural or computational overhead; as before, they simply express the utility of various external world states, and the arbiter exercises more control over which states may be achieved and how. This means of jointly determining turn and speed commands has not yet been implemented at this time.

It is also possible to place arbiters in nested control loops, with a map-based utility arbiter providing inputs to an effect arbiter which in turn feeds an actuation arbiter whose output is used by a constraint arbiter. For example, the path arbiter and turn arbiter could be nested as shown schematically in Figure 4-17. The path arbiter performs evidence fusion and evaluates candidate actions, sending the results as votes to the turn arbiter which performs command fusion and selects an action to be executed, and sends that turn command to the path arbiter as feedback.

![Diagram of multiple arbiter nested control loop.](image)

**Figure 4-17**: Multiple arbiter nested control loop.
Path arbiter sends evaluations as votes to turn arbiter and receives turn commands as feedback.

## 4.2 Behaviors

Within the framework of DAMN, behaviors must be defined to provide the task-specific knowledge for the domain. These behaviors are not part of the DAMN architecture *per se*. Each behavior runs completely independently and asynchronously, using whichever sensor data and processing algorithms are best suited to the task, and providing votes to the arbiter each at its own rate and according to its own time constraints.

DAMN is designed so that various behaviors can be easily added or removed from the system, depending on the current task at hand. Although the modules described below all use very different paradigms and representations, it has been relatively straightforward to integrate each and every one of them into the framework of DAMN. Sensor fusion is not necessary since the command fusion process in DAMN preserves the information that is critical to decision-making, yielding the capability to concurrently satisfy multiple objectives without the need for centralized...
bottlenecks. All of the behaviors described have been used in conjunction with each other in various configurations, yielding systems that were more capable than they would have been otherwise.

### 4.2.1 Evolutionary System Development

Conceptually, three levels of competence [13] have been implemented in DAMN thus far, as shown in Figure 4-18. These levels of competence are convenient for describing the incremental manner in which the system’s capabilities evolve; however, it is important to note that all behaviors co-exist at the same level of planning. The importance of a behavior’s decisions is reflected by the weighting factor for its votes, and is in no way affected by the level of competence in which it is described. As new functions are needed, additional behaviors can be added to the system without any need for modification to the previously included behaviors, thus preserving their established functionality and allowing for evolutionary development.

The safety behaviors form a first level of competence upon which other levels can be added. In contrast to priority-based architectures which only allow one behavior to be effective at any given moment, the structure of DAMN and its arbitration scheme allow the function of these safety behaviors to be preserved as additional levels of competence are added. Movement is the second level of competence that has been implemented; road following, cross-country navigation, and teleoperation behaviors have all been run together with the obstacle avoidance behavior to provide various forms of generating purposeful movement while maintaining safety [73]. The third level of competence is comprised of the various map-based goal-seeking behaviors. Cross-country behaviors have been combined with goal-oriented behaviors to produce directed off-road navigation [34] [70]. These behaviors are described in more detail in the sections that follow.

![Levels of competence in DAMN.](image)

Figure 4-18: Levels of competence in DAMN.
4.2.2 Safety Behaviors

Obstacle Avoidance Behaviors

An important class of behaviors in the context of vehicle safety are the Obstacle Avoidance behaviors, which detect obstacles and decide vehicle and sensor actions based on their location. The obstacles may be defined as traversable regions of terrain determined by range image processing [18] [28] or by stereo vision [46], by sonar detection of objects above the ground plane [39], or any other means of obstacle detection that is appropriate to the current task and environment. The obstacle avoidance system used most extensively with DAMN has been SMARTY [40], which has been applied to instantiate the turn, speed, and field of regard behaviors described below.

AVOID OBSTACLES (TURN BEHAVIOR)

The AVOID OBSTACLES behavior sends turn votes in favor of those directions in which the vehicle may safely travel. In order to decide this, it constructs a local map of obstacles in vehicle-centered coordinates and evaluates each of the possible command options, as illustrated in Figure 4-19. If a trajectory is completely free of any neighboring obstacles (such as the Straight Ahead or Hard Right turns shown in Figure 4-19), then the AVOID OBSTACLES behavior votes for travelling along that arc. If an obstacle lies in the path of a trajectory, the behavior votes against that arc, with the magnitude of the penalty proportional to the distance from the obstacle. Thus, the behavior votes more strongly against those turns that would result in an immediate impact (Hard Left in the figure) and votes less strongly against those turns which would only result in a collision after travelling several meters (Soft Right). In order to avoid bringing the vehicle unnecessarily close to an obstacle, the behavior also votes against those arcs that result in a near miss (soft left), although the evaluation is not as unfavorable as for those trajectories leading to a direct collision.

SLOW FOR OBSTACLES (SPEED BEHAVIOR)

The SLOW FOR OBSTACLES speed behavior operates in conjunction with the AVOID OBSTACLES turn behavior described above. As illustrated in Figure 4-19, the behavior constructs a map of
current obstacles in vehicle-centered coordinates. The behavior receives as additional input the turn radius selected by that arbiter and the current vehicle turn radius reported by the controller, in the manner shown in Figure 4-14. Instead of evaluating all of the arcs which may be commanded by the turn arbiter, the behavior decides a safe speed based only on the arcs corresponding to these current and future vehicle trajectories. If a trajectory is completely free of any neighboring obstacles, then the behavior votes for going at the maximum speed, which is a parameter specified at initialization. If an obstacle lies in or near the chosen arc, then the behavior votes to go slower as the distance from the obstacle decreases, until at some point it indicates that the vehicle must stop.

**AVERT OBSTACLES (FIELD OF REGARD BEHAVIOR)**

The **AVERT OBSTACLES** field of regard behavior uses the local map of detected obstacles to vote against looking in the direction of known obstacles; since the vehicle will not be travelling in that direction, there is no immediate benefit in viewing it. For example, when the map indicates that there are obstacles to the right of the vehicle, as in Figure 4-20a, the behavior votes in favor of a field of regard to the left as shown in Figure 4-20b where gray scales indicate the relative vote values and the darkest is the most favored. The plot of vote value versus pan angle in Figure 4-20c shows the series of votes in favor of panning to the left.

![Figure 4-20](image)

**Figure 4-20:** Avert Obstacles field of regard voting.

a) local map indicates obstacles to right of vehicle, so b) behavior votes in favor of field of regard polygons to left, resulting in c) plot of vote versus pan angle indicating to pan to left.

**AVOID OBSTACLES (UTILITY BEHAVIOR)**

When the **AVOID OBSTACLES** behavior is used in conjunction with the map-based utility arbiter, it simply sends obstacle locations to that arbiter. The behavior does not need to evaluate trajectories, and does not even need to know what type of vehicle, actuation, or controller is being used. For each obstacle detected, the behavior sends to the arbiter a large negative utility with a small standard deviation associated with the obstacle *per se* to avoid collision. Another negative utility with a smaller value and larger standard deviation is also assigned to the obstacle location to reflect the problems associated with getting too close to an obstacle, i.e., constrained mobility, occlusion of unknown areas, etc. The obstacle can be represented as either a point or as a polygon with known extent.
Another vital aspect of vehicle safety is insuring that the commanded turn stays within the dynamic constraints of the vehicle as it travels over varying terrain conditions at different speeds. The most important of these constraints is the one that insures that the vehicle will not tip over. Given a velocity of magnitude \( v \), the maximum positive and negative curvatures \( \kappa \) to avoid tip-over would be:

\[
\pm \kappa_{max} = \frac{\pm (\eta \cdot g \cdot \cos \rho) + g \cdot \sin \rho}{v^2}
\]

where \( \eta \) is the ratio of the distance between the vehicle’s center of gravity (c.g.) and the wheels to the c.g. height, \( g \) is the acceleration due to gravity, and \( \rho \) is the vehicle roll with respect to the gravity vector, as illustrated in Figure 4-21.

![Figure 4-21: Vehicle dynamics variables.](image)

Similar constraints can be imposed on vehicle the turn radius in order to avoid tire slippage. The limit on curvature for slippage is:

\[
\pm \kappa_{max} = \frac{(\mu \cdot g \cdot \cos \rho) \pm (g \cdot \sin \rho)}{v^2}
\]

where \( \mu \) is the dynamic coefficient of friction between the tire and the terrain. The LIMIT TURN behavior implement these two constraints, sending votes to the arbiter against commands that violate them.
**LIMIT SPEED (SPEED BEHAVIOR)**

As with curvature, the commanded speed must stay within the dynamic constraints of the vehicle as it travels over varying terrain conditions. Given a velocity of magnitude \( v \) and a chosen curvature \( \kappa \), the maximum velocity is:

\[
 v_{\text{max}} = \min \left( \pm \left( \frac{\eta \cdot g \cdot \cos \rho + g \cdot \sin \rho}{\kappa} \right)^{1/2} \right) \tag{Equation 4-19}
\]

where \( \eta, g, \) and \( \rho \) are as defined above. With a dynamic coefficient of friction of \( \mu \) between the tire and the terrain, the limit on speed to avoid tire slippage is:

\[
 \pm v_{\text{max}} = \left( \frac{(\mu \cdot g \cdot \cos \rho) \pm (g \cdot \sin \rho)}{\kappa} \right)^{1/2} \tag{Equation 4-20}
\]

The LIMIT SPEED behavior sends maximum speeds to the arbiter based on these two constraints.

**LIMIT PAN (FIELD OF REGARD ARBITER)**

Rather than a behavior sending votes to the arbiter based on system limits, LIMIT PAN is implemented within the Field of Regard arbiter and is used to inform behaviors which field of regard polygons can be immediately panned to. This determination is based on the current camera pan and tilt angles, vehicle movement, the arbiter’s cycle time, and the velocity and acceleration limits of the camera’s pan/tilt mechanism. This information may be used by Field of Regard behaviors to avoid evaluating polygons that cannot be viewed given the constraints of the system.

**LIMIT TURN (UTILITY ARBITER)**

The utility arbiter considers the vehicle’s dynamic constraints when determining which trajectories are to be evaluated, therefore no behavior is needed to represent them.

### 4.2.3 Movement Behaviors

**Road Following**

Once vehicle safety has been assured by the obstacle avoidance and dynamic constraint behaviors, it is desirable to add additional behaviors that provide the system with the ability to achieve the tasks for which it is intended. One such capability is road following; two behaviors have been implemented within DAMN that provide this function.

**SCARF (TURN BEHAVIOR)**

The Supervised Classification Applied to Road Following system (SCARF) uses a color classification scheme to identify regions of terrain as road and non-road [17]. Once colors in the
image have been clustered and labeled, a road template is applied to determine the probability that the road center lies at each point in the subsampled image, and a Hough space of road position and orientation is created. SCARF was designed to run in isolation; the image point with the highest such probability is selected and the vehicle is commanded to drive towards the corresponding point on the ground plane using the pure pursuit algorithm [76].

In order to integrate SCARF with in the framework of DAMN, its output must be modified so that it issues a series of turn command votes as described above. The most straightforward means of accomplishing this would be to simply take the steering direction command output and convolve it with a Gaussian, resulting in turn votes of the form shown for the behaviors in Figure 4-2 (a & b). However, in the interest of fully exploiting the information available within a module, a behavior was designed that accepted as input the Hough transform used as an intermediate representation within SCARF. This behavior’s processing of this input is illustrated in Figure 4-22: for each candidate turn command, the point at a fixed lookahead distance as used in the pure pursuit algorithm is located, and the maximum probability for all roads whose center line passes through that point is used as the vote value for that turn. Thus, each arc is evaluated independently, so that if two roads are located within the image, for example, the SCARF behavior would vote favorably for both of them, leaving the arbiter to decide which road to follow in conjunction with information from other behaviors.

![Figure 4-22: Evaluating a turn radius within SCARF.](image)

**ALVINN (TURN BEHAVIOR)**

The ALVINN road following system is an artificial neural network that is trained, using backpropagation, to associate preprocessed low resolution input images with the appropriate output steering commands [57]. In the case of ALVINN, creating a behavior that independently evaluated each arc was relatively straightforward. The units of the neural network’s output layer each represent an evaluation of a particular turn command, with the layer trained to produce Gaussian curves centered about those turns that would follow the road ahead. These units are simply resampled to the DAMN voting command space, using a Gaussian of the appropriate width. This process is illustrated in Figure 4-23.
Teleoperation is another possible mode in which a robotic system may need to operate. The STRIPE teleoperation system [33] provides a graphical user interface allowing a human operator to designate waypoints for the vehicle by selecting points on a video image and projecting them on to the surface on which the vehicle is travelling. STRIPE then fits a spline to these points and uses pure pursuit to track the path. When used in isolation, it simply sends a steering command to the controller; when used as a DAMN behavior, it sends a series of votes representing a Gaussian centered on the desired command. This allows the dynamic constraints and obstacle avoidance behaviors to be used in conjunction with STRIPE so that the safety of the vehicle is still assured.

4.2.4 Goal-Directed Behaviors

Subgoals

Another important level of functionality that should be present in any general purpose robotic system is the ability to reach certain destinations using whatever global information is available. While the low-level behaviors operate at a high rate to ensure safety and to provide functions such as road following and cross-country navigation, high-level behaviors are free to process map-based or symbolic information at a slower rate, and periodically issue votes to the arbiter that guide the robot towards the current goal.

SEEK GOALS (TURN BEHAVIOR)

The SEEK GOALS turn behavior is one way to provide this capability. This behavior directs the vehicle toward a series of goal points specified in global coordinates either by the user [39] or by a map-based planner [34]. Pure pursuit [76] is then used to generate turn commands that track a
line between consecutive subgoals. The desired turn radius is transformed into a series of votes by applying a Gaussian whose peak is at the desired turn radius and which tapers off as the difference between this turn radius and a prospective turn command increases. A goal is considered satisfied once the vehicle enters a circle centered at the goal location; then the next goal is pursued. Because of errors in goal placement and accumulated errors in vehicle positioning, a goal point may not be reachable. For this reason, an ellipse is defined with the current goal and the subsequent goal as foci; if the vehicle enters this ellipse, the current goal is abandoned and the next one becomes the current goal instead, thus allowing progress to continue.

**FOLLOW PATH (UTILITY BEHAVIOR)**

The **FOLLOW PATH** behavior associates a positive utility with each of the subgoals to be reached by the vehicle, shown as a point with circles of increasing uncertainty in Figure 4-24. A positive line utility between subgoals is also defined, along with a one-dimensional gaussian so that a corridor is defined between consecutive goals. As seen in the figure, vehicle trajectories accumulate greater positive utility the closer they are to the line between goals, thus keeping the vehicle as close to that line as possible without the need to perform explicit pure pursuit calculations. Thus, the vehicle tends to stay within the corridor between goals, even in the presence of negative utilities due to factors such as obstacles. Unlike the **SEEK GOALS** turn behavior, which must monitor vehicle progress and decide when a goal has been achieved or should be abandoned, this behavior can simply send all utilities to the arbiter at once, and each attracts the vehicle in turn as it gets closer; if a subgoal is unreachable or inadvertently bypassed, then the utilities defined by the next corridor attract the vehicle to the next subgoal.

![Figure 4-24: Follow Path behavior.](image)

Positive point utilities correspond to subgoals, line utilities between subgoals create corridors.

**Dynamic Programming**

Some more sophisticated map-based planning techniques have also been integrated and used within the DAMN framework. These planners use dynamic programming techniques based on the A* search algorithm [49] to determine an optimal global path. However, an important point is that they do not hand a plan down to a lower level planner for execution, but rather maintain an internal representation that allows them to participate directly in the control of the vehicle based on its current state. A* yields a set of pointers within the map grid that point toward the goal, as depicted by the small arrows in Figure 4-25. During execution, this grid may be indexed by the current vehicle position to yield a path towards the goal which is optimal based on the information available in the map at that time.
**FOLLOW GRADIENT FIELD (TURN BEHAVIOR)**

The Internalized Plans [53] approach uses a detailed map to perform an A* search from the goal(s) back toward the start point to create a “Gradient Field” towards the goal. The type and slope of the terrain, among other factors, is used to estimate the cost of traversal between grid cells. During run-time, the grid cell containing the current vehicle location is identified, and the Gradient Field pointers are followed forward to the point $G'$ in Figure 4-25; the desired heading to reach the goal is that from the current location $S$ to $G'$, and a series of votes with its peak at that value is sent to the turn arbiter.

![Figure 4-25: Following Gradient Field pointers.](image)

**DECREASE GOAL DISTANCE (TURN BEHAVIOR)**

The D* planner [69] also creates a grid with “backpointers” that represent information on how best to reach the goal from any location in the map. The map may initially contain no information, but is created incrementally as new information becomes available during the execution of a mission, and the arc traversal costs and backpointers are updated to reflect this new knowledge. The resulting global plan is integrated into DAMN as a behavior by determining, for each possible turn command, the weight $w$ of reaching the goal from a point along that arc a fixed distance ahead (the squares designated collectively as $S'$ in Figure 4-26). If $w_{\text{max}}$ and $w_{\text{min}}$ are the maximum and minimum values of $w$, then the vote for each turn command is determined as: 

$$w_{\text{max}} - w \, , \, (w_{\text{max}} - w_{\text{min}}) \div (w_{\text{max}} - w_{\text{min}}) \, , \, (w_{\text{max}} - w) \, , \, (w_{\text{max}} - w_{\text{min}}) \div (w_{\text{max}} - w_{\text{min}})$$

In the case that a point $S'$ is not represented on the grid, or if the goal cannot be reached from it, then the vote for that arc is set to -1.

![Figure 4-26: Using D* to evaluate distance from goal for each arc.](image)

**DECREASE GOAL DISTANCE (UTILITY BEHAVIOR)**

In the case where D* is used to implement a behavior that communicates with the map-based utility arbiter, there is no need to perform arc evaluation. Instead, the behavior simply sends the
immediate portion of its map as a grid to the arbiter, and sends a new portion of the map as information is updated or as the vehicle approaches a new region of the map. Only a small portion of the map needs to be sent to the arbiter, and updates need not occur often, so that the bandwidth requirements of this behavior are kept within a very reasonable level.

Each cell in the D* map has an estimated distance to the goal point, $h_c$. Let the estimated goal distance from the current vehicle position be $h_0$. We then obtain the estimated utility of a cell by taking the difference of the two: $u_c = h_0 - h_c$. This gives a positive value when the cell’s goal distance is less than the current distance to the goal, negative when the distance is greater, and zero when there is no change in the distance to the goal. This results in grids sent to the arbiter like that shown in Figure 4-27. The center column of the bottom row corresponds to the grid’s associated vehicle position, sent to the arbiter in conjunction with the grid. The grid’s resolution and dimensions are specified by the behavior when it first connects to the arbiter. The lighter stripes at the bottom of the grid indicate small positive utilities, and they go from lighter to darker as the positive utility increases at the top of the grid, i.e. as the distance to the goal decreases. In this example D* has also been notified of a detected obstacle and placed it in the map; this appears in the grid as a dark (red) circle surrounded by a lighter (orange) ring indicating the presence of a large negative utility.

![Figure 4-27](image)

**Figure 4-27**: Grid-based utility based on distance to goal.
Light to dark stripes indicate increasing positive utility, dark circle indicates large negative utility.

**LOOK TOWARDS GOAL (FIELD OF REGARD BEHAVIOR)**

This behavior votes to look at field of regard polygons in the direction of travel indicated by the map-based plan, thus encouraging the system to create maps and move towards the goal.

**EXPLORE UNMAPPED TERRAIN (FIELD OF REGARD BEHAVIOR)**

The D* map-based planner needs to acquire sensed images in order to incrementally create a map. So that the system may expand its map, a separate behavior module within D* votes to view terrain that has not been mapped yet but is contiguous to terrain that has been mapped.
Various other auxiliary behaviors that do not achieve a particular task but issue votes for secondary considerations may also be run. These include the DRIVE STRAIGHT behavior, which simply favors going in whatever direction the vehicle is already heading at any given instant, in order to avoid sudden and unnecessary turns, and the MAINTAIN TURN behavior, which votes against turning in directions opposite to the currently commanded turn, and which helps to avoid unnecessary oscillations in steering. These behaviors are unnecessary for the map-based utility arbiter; their functionality is provided internally by the arbiter biases, as described in Step 4 of the “Operation of Arbiter” subsection in Section 4.1.4.

4.2.5 Combining Behaviors

DAMN is designed so that various behaviors can be easily added or removed from the system, depending on the current task at hand. Although the modules described above all use very different paradigms and representations, it has been relatively straightforward to integrate each and every one of them into the framework of DAMN; some of these behaviors were implemented without any assistance from me, and in some cases even without my knowledge until after they had been successfully used in system experiments!

The voting strengths, or weights, of each behavior are specified by the user, and are then normalized by the arbiter so that their sum equals 1. Because only the relative values are important, and because the magnitude of each behavior's votes vary according to their importance, DAMN is fairly insensitive to the values of these weights and the system performs well without a need to tweak these parameters. For example, the AVOID OBSTACLES behavior has been run in conjunction with the SEEK GOAL behavior with relative weights of 0.75 and 0.25, respectively, and with weights of 0.9 and 0.1, and in both cases has successfully reached goals while avoiding obstacles.

The vote weights of each behavior can also be modified by messages sent to the arbiter from a mode manager module. It can reconfigure the weights according to whatever top-down planning considerations it may have, and potentially could use bottom-up information about the effectiveness and relevance of a behavior [55]. Different modes of operation that exclude some behaviors can be constructed by setting the weights those behaviors to 0. A Mode Manager was developed at Hughes to be used with DAMN for this purpose, and at CMU Annotated Maps were integrated with DAMN to provide this capability [73].

4.3 Development Tools

A crucial consideration in the design of a mobile robot architecture is the ease with which a system may be developed, tested, debugged, and understood. In addition, it should not be overly restrictive so that a wide variety of independently developed modules may be integrated within its structure. It must provide a framework for sensing and reasoning processes to be conducted in a
timely fashion within the context of purposeful, goal-oriented behavior. While many general purpose architectures are computationally equivalent or nearly so, the ease with which various classes of systems may be instantiated within them varies greatly.

One of the primary goals in constructing DAMN was to make it as simple as possible to take an independently developed module and hook it in to a complete system by adding the appropriate interface and without having to make substantial changes in the operation of that module. By virtue of its capability of integrating additional modules without any need to modify or suppress existing behaviors, DAMN provides a framework for evolutionary development of robot systems.

### 4.3.1 Creating Behaviors

DAMN is designed to be an open architecture where new behaviors can easily be added to the system as the need arises. To facilitate this process, some tools for creating behaviors and communicating with their respective arbiters are provided. This may be either through object-oriented programming or library techniques.

**Behavior Object Hierarchies**

To facilitate the development and maintenance of behaviors, maximize the reuse of code, and ensure that new behaviors adhere to the framework for which they are defined, an object-oriented class hierarchy has been developed, as shown in Figure 4-28. These classes, implemented in C++, standardize inter-module communication (using IPT [25]), data collection, the arbiter voting process, process I/O, data recording, parameter initialization and various other loop constructs. Multiple inheritance is used for those behaviors that issue votes to more than one arbiter construct.

![Behavior module class hierarchy.](image)

**Figure 4-28**: Behavior module class hierarchy.
The behavior classes contain all the common functionality necessary for operation, and the individual details of each behavior’s function are filled in by defining or adding to the following member functions:

- register_messages()
- initialize_connections()
- request_data()
- send_data()

In addition, the functions are provided to the behaviors which they may use as needed within their body. For example, the functions provided to an object derived from the turn behavior class are:

- get_turn_vote(index)
- set_turn_vote(index, value)
- set_all_turn_votes(value)
- rescale_votes()
- vote_gaussian_curvature(mean_curvature, std_dev)

Arbiter Interface Libraries

One of the main constraints and goals of DAMN is that it must be able to integrate subsystems that were developed outside the context of the architecture, with only minor additions or modifications necessary. Thus, while developing behaviors within the object-oriented module hierarchy may be highly desirable from a software engineering standpoint, it is not realistic nor desirable to demand that all behaviors be developed in this manner. In order to easily integrate modules developed outside the framework of DAMN, a set of ‘C’ interface libraries are available which provide much of the same functionality as the behavior objects. This interface was kept as simple as possible, hiding details of the arbiter’s operation whenever possible. Separate libraries are provided for each arbiter; behaviors wishing to interact with multiple arbiters need simply link with the appropriate set of libraries. The functions provided by the turn arbiter interface library for use by behaviors are:

- DAMN_behavior_init_comm(behavior_name, communicator)
- DAMN_behavior_get_turn_vote(index)
- DAMN_behavior_set_turn_vote(index, value)
- DAMN_behavior_rescale_turn_votes()
- DAMN_behavior_vote_gaussian_curvature(mean_curvature, std_dev, peak_vote)
- DAMN_behavior_send_turn_votes()
• DAMN_behavior_send_turn_limit(curvature_limit)
• DAMN_behavior_get_arc_curvature(index)
• DAMN_behavior_get_arc_index(curvature)

4.3.2 User Interface

A simple text interface which optionally outputs informational messages from each running module is provided for debugging and logging purposes. The user may also interactively start and halt the arbiters and vehicle, vary parameters such as maximum speed, and toggle debugging output and data recording.

A Graphical User Interface*, shown in Figure 4-29, has also been integrated into the DAMN turn arbiter. It outputs the votes issued by each active behavior, as well as their current weights. The arbiter’s combined votes and chosen turn command are also displayed. The user may also use this interface to modify behavior weights, thus providing meta-level control.

![Figure 4-29: Graphical User Interface for turn arbiter.](image)

The map-based utility arbiter also displays the utilities received from behaviors and the expected utility for each evaluated arc, as shown in Figure 4-30. A color display is used, and negative utilities are shown as shades of red, orange and yellow, while positive utilities are shown as shades of blue and green. Figure 4-30a shows a series of negative point utilities indicating the presence of detected obstacles and the path which the vehicle has taken to avoid them. Figure 4-30b shows a line of positive utility ending in a goal point with positive utility, with three

* Developed by Alan Dickinson at Martin Marietta
negative point utilities along the way; concentric circles indicate standard deviations of the
gaussian uncertainty associated with the point. Figure 4-30c shows an array of votes received
from a map-based behavior which uses grid voting rather than points or polygons; the blue/green
background indicates small positive utilities and the dark red areas surrounded by a lighter orange
border indicate the presence of large negative utilities in the grid.

![Figure 4-30 : Map-based utility arbiter display.](image)
a) negative point utilities and vehicle path, b) line utility and point utilities with concentric circles
indicating gaussian uncertainty, c) grid votes with dark areas indicating large negative utility.

There also exist behaviors which accept commands from the user so that he may directly affect
vehicle control via the voting process. This behavior input is combined by the arbiter with all
other active behaviors’ votes to determine the ultimate choice of vehicle commands. The user can
vote for an arc turn command which is convolved with a gaussian to create a set of votes sent to
the turn arbiter. The use can also send speed commands, either by entering an absolute number or
varying the relative speed with keyboard input. Depending upon the type of speed arbitration
being used, this may directly determine the vehicle speed, set the maximum speed multiplication
factor, or set an upper bound which may be lowered by other behaviors. The user may also
command the pan and tilt of the camera, or vote for a field of regard polygon. The user may also
enter pointed, linear, or polygonal utilities in the map with specified values and optional gaussian
standard deviations.

4.3.3 Data Recording and Playback

It is important to record system data so that it may later be used for analysis, display, and most
importantly debugging. Recording all data from all behaviors would be a very difficult thing to do
since the nature of their input and processing is unconstrained, and because all behaviors operate
asynchronously. It would be problematic to attempt the synchronization of internal clocks for all
processes on multiple processors for the purpose of determining a complete order among all
recorded events. Instead, it was decided that all data would be recorded by the centralized arbiter.
In any case, individual modules are developed independently and can be debugged in stand-alone
mode since they are not required to use DAMN and are usually developed as stand-alone first
anyhow. Upon integration, the data from the arbiter can then be used to analyze the interaction between modules.

All arbiters optionally output data to the user and record it to a log file. The data recorded includes votes or utilities received from the behaviors, the results of combining these votes, the commands sent to the controller, and controller data such as vehicle position, heading, speed, and turn radius. All data is also tagged with the time at which it is written to the log file. The data in these log files are written as text so that they may be directly viewed by the system developer. Programs also are provided for computing metrics such as path length, duration, smoothness, average distance from obstacles, etc., and for displaying the data using a plotting package; these statistics and displays are evident throughout Chapter 5.

A playback facility also exists to recreate the recorded events. This is extremely useful for debugging purposes as the system developer can recreate the vehicle run, watching events unfold in the order in which they were noted by the command arbiter. A display like that shown in Figure 4-30 is created, and textual information may also be displayed if desired. The playback can occur in real-time, or may be sped up or slowed down. In addition, the playback facility shares a command arbitration library with the appropriate arbiter, so that at any point in the playback the developer may request that the current votes be arbitrated and all intermediate computations are displayed. This is also useful to execute from within a debugger so that the step by step execution of the process may be scrutinized.

## 4.4 Conclusion

The details of how the various type of implemented arbitration schemes were provided. A speed arbiter was implemented using the constraint arbitration scheme, the simplest. A turn arbiter accepts and arbitrates votes in a curvature-based actuation space whose results are sent directly to the vehicle controller. Effect arbitration allows behaviors to send votes to a field of regard arbiter rather than directly choosing pan and tilt angles, thus allowing for more coherent reasoning involving the motion of the vehicle system. A map-based utility arbiter uses evidence fusion as a new means of action selection that allows the arbiter to compensate for the kinematics and dynamics of the system and to effectively synchronize the outputs of the behaviors.

Command spaces in DAMN may be directly at the actuator level, or abstracted into desired effects of actuation. These command spaces are partitioned into discrete action choices, thus allowing the approximate representation of any arbitrary voting function. Although the internal command representations used by the arbiters are discrete, the inputs to them from the voting behaviors may be continuous. The inputs to the map-based arbiter are not commands but utilities; behaviors determine desired world states rather than desired actions. The map-based arbiter internally reasons in a discrete actuator command space, but its interface consists of maps that may be either continuous or discrete. The map-based utility space is not time-dependent, so that an arbiter using such a representation is capable of effectively synchronizing and maintaining a
consistent interpretation of the votes received from asynchronous behaviors, thus providing coherent reasoning in a distributed system.

The implementation of several behavior was described. Behaviors may be completely reactive or deliberate at whichever level is appropriate to the task, and send their output to the arbiter asynchronously. Behaviors that avoid obstacles and ensure that the dynamic constraints of the system are not violated comprise a first level of competence upon which other levels can be added. Road following, cross-country navigation, and teleoperation behaviors form the second level of competence that provides the system with basic navigational abilities. Goal-based behaviors using subgoals or dynamic programming provide the third level of competence. Combinations of all these different types of behaviors have been integrated with DAMN using the various types of arbiters, creating complete navigational systems that have successfully been used for various purposes [34] [70] [73].
5.1 Performance Metrics

A fundamental capability which a robot control architecture must provide is the ability to avoid obstacles. Therefore, the average distance between the vehicle and obstacles in its path is an important metric to be considered when evaluating the path taken. Another important quality for a robot navigation system is the ability to follow a path to reach a goal point, so a second metric used to judge the quality of an executed path is the average distance between the vehicle and the current goal. Because the significance of relative distances is greater when those distances are small, the square of distance is used in both of the above metrics.

Another general measure of the quality of the vehicle path is its smoothness [30]. While it may not always be possible to control the vehicle in a smooth manner, for example when avoiding obstacles, it is nonetheless a desirable trait that should be considered in judging the quality of a path and of the system that commanded it. All else being equal, the smoothness of a path reflects consistency in decision-making and the ability to anticipate and respond to events in a timely fashion.

These three metrics are of course not always compatible; one must be sacrificed for the other. For example, if a vehicle were to always avoid obstacles by the greatest possible distance, as if following a Voronoi edge, then path smoothness may suffer and the vehicle may stray unnecessarily far from goals it is trying to reach. Smoothness would be maximized by always following a straight line or going in a circle of constant curvature. Generally speaking, however, these metrics may be ranked into order of importance: mean obstacle proximity clearly being the most important, and then mean goal distance and path smoothness follow. These three metrics could be combined into one larger objective function by taking a weighted sum of their measures. The analysis here instead explores each in isolation in order to more clearly determine the effects of manipulating certain system variables.

These three performance metrics, mean obstacle proximity, mean goal distance, and path smoothness, are each obtained by integrating values along the entire path and normalizing. They are defined below in such a way that each is meant to be minimized; a lower value implies a better path from the perspective of that objective.

5.1.1 Mean Obstacle Proximity

An important metric for a vehicle path is the average distance to obstacles along that path. Considering obstacle avoidance in isolation, it would be most desirable to maximize the distance between the vehicle and any obstacles surrounding it. The distance to that obstacle which is closest to the vehicle at any given moment provides a measure of safety clearance; when inverted, it provides a measure of proximity to obstacles which is to be minimized. The square of distance is used to reflect the increasing relative importance of obstacle proximity when the vehicle is
closer to an obstacle. Thus, the mean obstacle proximity metric for a path is defined by the inverse square of the distance \( l_o \) to the closest obstacle, integrated along the path and normalized by the total number of path points \( n \):

\[
\text{mean obstacle proximity} = \frac{\int \left( \frac{1}{l_o} \right)^2 ds}{n}
\]

Lower mean obstacle proximity means that the vehicle was on the average further away from the nearest obstacle, and therefore the path was better in terms of maintaining a safety clearance.

### 5.1.2 Mean Goal Distance

This measure is analogous to mean obstacle proximity, except we want to minimize the distance to a goal rather than minimize proximity, so no inverse is taken. The mean goal distance of a path is defined by the square of the distance \( l_g \) to the closest goal, integrated along the path and normalized by the total number of path points \( n \):

\[
\text{mean goal distance} = \frac{\int l_g^2 ds}{n}
\]

Lower mean goal distance means that the vehicle was on the average closer the nearest goal, which in the case of the follow heading behavior means closer to the line of positive utility.

### 5.1.3 Smoothness

Smoothness is defined by the square of the change in vehicle curvature \( \kappa \) with respect to time, integrated along the path and normalized by the total time \( t \):

\[
\text{smoothness} = \frac{\int \left( \frac{d\kappa}{dt} \right)^2 ds}{t}
\]

A lower smoothness measure means that curvature changed less over the course of the path, and therefore that the vehicle path was smoother.
5.2 Vehicle and System Characteristics

5.2.1 Vehicle Dynamics

Steering Actuator Response

To determine the vehicle’s steering response characteristics, simple tests were performed with the vehicle parked, issuing commands and recording the steering curvature at a high sampling rate. The empirical data shown in Figure 5-1 for the Navlab 2 HMMWV vehicle shows that, after an initial delay $\delta$, the steering actuator rotates the wheels linearly with time. A linear regression analysis of this data yields an x-intercept value of 0.20 seconds for $\delta$ and a slope of 0.13 radians/seconds$^2$ for $k$, the constant rate of change of curvature. Thus, the vehicle path initially follows a clothoid pattern $\kappa = \kappa_0 + k(s)$ as described in Section 3.4.2, “Control Issues,” on page 50. Once the target curvature has been achieved, then the vehicle follows an arc of constant radius.

![Vehicle Steering Response](image)

Number of observations = 50
Correlation coefficient = 0.99
Regression coefficient ($\kappa$) = 0.13 radians/seconds$^2$
X-intercept ($\delta$) = 0.20 seconds

Figure 5-1: Steering actuator response characteristics.

Controller Response Latency

The data indicated in the figure above indicates that the delay between the time a command is issued and the time it begins to be executed is 0.2 seconds, which was verified by data collected during actual vehicle runs. The solid line in Figure 5-2a shows commanded curvature as a function of the time it was sent to the controller, and the dashed line indicates the actual vehicle
curvature as a function of the time it was reported by the controller. Closer inspection of these functions confirms delay of 0.2 seconds between the time a curvature is commanded to the controller and the time at which it is executed, which can be readily observed in the close-up plots of this same data in Figure 5-2b and Figure 5-2c; the latter also shows that the controller cannot track high frequency changes in commanded curvature, resulting in a smoothing effect.

5.3 Turn Arbiter: Vehicle Run Results

For the vehicle experiment detailed here, the DAMN system was configured as shown in Figure 5-3. An ERIM laser range finder generated a polar map of sensed depth every 500ms. This range image was then processed to create a Cartesian depth map which was then used to determine the presence and location of obstacles, a process that took approximately 300ms to complete [39]. These obstacle locations were in turn used by the OBSTACLE AVOIDANCE turn
behavior to update a local map of the area around the vehicle [40]; arcs were then evaluated within that map to generate turn votes in the manner described in Section 4.2.2, “Safety Behaviors,” on page 86. This process might take as little as 50ms, depending on the number of obstacles detected, but regardless votes were only sent to the turn arbiter once every 100ms in order to avoid limitations imposed by the communications system in use at that time [21].

![Figure 5-3](image_url) : Turn arbiter with Obstacle Avoidance and Seek Goals behaviors.

The SEEK GOALS behavior, described in Section 4.2.4, “Goal-Directed Behaviors,” on page 91, received as input a series of goal points specified by the user, which were on average 100 meters apart. Votes were generated that favor turn commands that track a line between consecutive goal points. As the vehicle came within eight meters of the current target goal point, the next one was pursued in turn until the final goal is achieved. The processing requirements for this behavior were minimal, but again votes were sent to the turn arbiter at a rate of once every 100ms.

The weights for the Obstacle Avoidance and Goal Seeking behaviors were 0.8 and 0.2, respectively. The obstacle weight was large relative to the goal weight to reflect the fact that avoiding obstacles is generally considered to be more important than approaching the goal at any given moment. The Turn Arbiter combined the votes from these behaviors and issued a new steering command every 100ms. The average vehicle speed was 2 meters/second in this experiment.

Figure 5-4 shows a typical run of the navigation system. Figure 5-4a shows the environment in which this experiment takes place; the white line superimposed on the image of the terrain shows the approximate path of the vehicle through this environment. Figure 5-4b shows a close-up view of one section of the loop. The terrain in the “slag heap” test area includes hills, rocks, and ditches. Figure 5-4c shows the actual path recorded during the experiment projected on the average ground plane, as well as the obstacle regions shown as black dots and the intermediate goal points shown as small circles.

Figure 5-5 illustrates more precisely the way the DAMN arbiter combined outputs from the different behaviors. The lower part of the figure shows the path of the vehicle, shown as a black line, around a set of obstacle points, shown as black dots. The upper part of the figure shows the distribution of votes for each of the behaviors and for the arbiter. The horizontal axis is the time axis, the total length of the sequence being 30 seconds. For each module, the vertical axis is the
400 meters
Five points of interest along the vehicle path in Figure 5-5, labeled as A through E, are shown in greater detail by the five graphs that follow in Figure 5-6. Each graph depicts the votes issued for each turn choice by the obstacle avoidance and goal behaviors, as well as the weighted sum of these votes as computed by the arbiter. The horizontal axis of each graph shows the possible turn radius choices encoded from -8 meters for hard left to +8 meters for hard right (curvature and turn radius are multiplicative inverses). The sequence of five points selected is typical of a path around a single obstacle.

![Figure 5-5: Distribution of votes over time in turn arbiter and behaviors.](image-url)

Chapter 5: Experimental Results
Figure 5-6: Vote distributions at five specified locations along vehicle path
At point A, the obstacle is first reported and the obstacle avoidance behavior generates high votes for turning left to go around the obstacle, and inhibits right turns with negative votes, as shown by the solid line in the graph. At the same time, the goal behavior’s vote distribution is relatively flat around the straight direction since the vehicle is currently headed in the desired direction; this is shown by the dashed line. Because of the small relative weight of the goal behavior, the combined vote distribution in the arbiter, shown as a thicker solid line, is dominated by the votes received from the obstacle avoidance behavior; a left turn is therefore commanded. At point B, the obstacle is still close to the vehicle and the votes distributions are similar to the ones at A, thus maintaining the vehicle to the left of the obstacle.

At point C, the obstacle avoidance behavior is still voting in favor of a sharp left turn, but the votes for the softer left turns is now not as low as it was at A or B, since the vehicle is now clearing the obstacles. At the same time, the goal behavior is starting to shift its votes towards turning right in order to bring the vehicle back to the target. The summed votes are still at a maximum for a left turn because of the greater weight of the obstacle avoidance behavior’s votes, and so the arbiter continues to steer the vehicle well clear of the obstacles.

By the time the vehicle has reached point D, it is just passing the obstacles, so that the obstacle avoidance behavior is now only disallowing extreme right or left turns because of field of view constraints. The goal behavior is now able to have more influence over the chosen turn direction, and a right turn is now executed so that the desired vehicle heading is restored. Note that between points C and D, there is a transition period where the turn radius chosen by the arbiter is neither the hard left favored by the obstacle avoidance behavior nor the medium right favored by the goal behavior. Instead, as turns other than hard left become acceptable to the obstacle avoidance behavior, and as the distribution of votes from the goal behavior shift further to the right, the vehicle is commanded to make ever softer left turns, and eventually turns to the right. As can be seen in the graph of the commanded turn radius in Figure 5-6, rather than abruptly switching modes from avoiding obstacles to following a heading, the transition proceeds smoothly and steadily as the situation gradually changes. Finally, at point E, the vehicle has completely cleared the obstacles and they are no longer in the local map, so that the votes from the obstacle avoidance behavior are mostly +1. The goal behavior now dominates completely and a right turn is commanded. Through the interaction of the avoid obstacles and goal behaviors, the vehicle was thus able to successfully circumvent an obstacle and regain its desired heading.

Unfortunately, the data for this path needed to generate the performance statistics described in Section 5.1 are not available; instead, a detailed analysis of the turn arbiter performance in simulation with respect to these metrics is provided below in Section 5.5. However, this experiment shows that the arbiter combines the preferred commands from different behaviors in an efficient and natural way, and that the resulting system is capable of successfully negotiating a path between a series of goal points while avoiding obstacles encountered along the way.
5.4 Path Arbiter: Vehicle Run Results

For the following vehicle experiments using the path arbiter, the system was configured as shown in Figure 5-7. As in the previous system, an ERIM laser range finder generated a polar map of sensed depth every 500ms. The map generation and obstacle detection functions were included within the OBSTACLE AVOIDANCE utility behavior, which was also responsible for local map maintenance as before; the combined behavior, which avoided communications delays, took approximately 250ms total processing time. At each iteration, the behavior sent utilities for all obstacles within the map, overriding ones previously sent to the arbiter so that the most recent information was always used. For each obstacle, one point utility with a value of \(-100000.0\) and a gaussian standard deviation of 0.2 and another with a value of \(-100.0\) and a standard deviation of 1.0; as described in Section 4.2.4, the former represents the danger of colliding with the obstacle, and the latter reflects the problems associated with getting too close to an obstacle, such as constrained mobility and occlusion of unknown terrain.

![Figure 5-7: Path arbiter with Obstacle Avoidance and Follow Path behaviors.](image)

The FOLLOW PATH behavior, described in Section 4.2.4, also received as input a series of goal points specified by the user, spaced 25 meters apart. It associates a positive utility of 10 with each of these subgoals, with a two-dimensional gaussian with standard deviation of 10 for each dimension. A positive line utility between subgoals is also defined with a utility of 25, with a one-dimensional gaussian of standard deviation 5 along it so that a corridor is defined between consecutive goals. The line utility was given a higher utility than the goal point so that the vehicle would be drawn back to the corridor when it strayed from it, rather than heading directly towards the subgoal If a subgoal is unreachable or inadvertently bypassed, then the utilities defined by the next corridor attract the vehicle to the next subgoal.

Unlike the SEEK GOALS turn behavior, which must constantly generate a new set of votes to direct the vehicle to the next goal, as well as monitor vehicle progress and decide when a goal has been achieved or should be abandoned, this behavior simply sends all utilities to the arbiter at once. Each goals attracts the vehicle in turn as it gets closer; if a goal is unreachable or inadvertently bypassed, then the utilities defined by the next corridor attract the vehicle to the next goal.

The Path Arbiter combined the utilities from these behaviors into its vehicle-centered utility map, evaluated candidate trajectories within that map, and issued the steering command that
maximized the expected utility. This process took 200ms on average, depending upon the number and type of utilities within the map.

5.4.1 Path Arbiter without Predictive Control

This experiment was run with the predictive control capability of the path arbiter turned off so that its effect could be observed and compared to the subsequent experiment conducted using predictive control. The experiment was conducted in the same “slag heap” test area used for the previous experiment. Figure 5-9a shows the path taken by the vehicle as it wound its way between the dense obstacle field indicated by the cross marks. The total length of the path was almost 400 meters, and the average vehicle speed was 0.7 meters/second. Figure 5-9b and Figure 5-9c show close-up snapshots of the vehicle in on-road and off-road portions of the path, respectively, as indicated by the arrows.

As can be seen in Figure 5-8, the vehicle oscillated quite a bit, yielding a smoothness measure which we will see is high compared to subsequent runs using predictive control. The metrics for this run are:

- Mean obstacle proximity = 0.41487
- Mean goal distance = 2882.7
- Path smoothness = 0.00432

![Figure 5-8: Vehicle curvature along on-road portion of vehicle run, w/o prediction.](image)
Figure 5-9: Vehicle run using path arbiter w/o prediction.
a) trace of full path, b) close-up of on-road portion of path, c) close-up of off-road portion of path.
5.4.2 Path Arbiter with Predictive Control

Further experiments were conducted with the path arbiter, this time with the predictive control capability in use. The previous experiment could not be duplicated due to several problems with the testbed vehicle and sensors. The laser range finder was replaced with a pair of stereo cameras, and depth map construction was sparser and more time consuming. In addition, the runs were conducted without a goal-based behavior in operation. In spite of these complications, significantly better results were obtained by using predictive control. Two separate short runs were made in the slag heap, as shown in Figure 5-10. The first run was 55 meters in length and the average speed was 0.9 m/s. The second path was 150 approximately meters long and the average speed was 0.8 m/s.

Figure 5-10: Path arbiter vehicle runs with predictive control.
The curvature profile for these runs is shown in Figure 5-11. It can be seen to be substantially smoother than the curvature generated without predictive control shown in Figure 5-8. This is borne out by the very low values of the smoothness metric for these runs, achieved while also maintaining a low obstacle proximity value. The mean goal distance metric was undefined for these experiments as no goal locations were specified.

**Figure 5-11**: Vehicle curvature for path arbiter with prediction.

Path Metrics for First Segment
- Path smoothness = 0.00007
- Mean obstacle proximity = 0.15770

Path Metrics for Second Segment
- Path smoothness = 0.00002
- Mean obstacle proximity = 0.14090
5.5 Controlled Simulation Experiments

A vehicle simulator was used for experiments were conditions could be carefully controlled and higher speeds could be used without risk of damage. A simulator was built which has the capability to mimic the two aspects of vehicle dynamics that we are interested in: system latency (i.e., lag between time a command is issued and time it begins to be executed), and actuator response time (i.e., rate at which control variable can change). The simulator does not attempt to reproduce the effects of noise in the position or control of the vehicle. Sensor noise was implicitly introduced by virtue of the fact that obstacle locations are taken from actual data processed during previous vehicle runs.

5.5.1 Experimental Design

The simulation testbed scenario used was as shown in Figure 5-12. The AVOID OBSTACLES behavior was run using obstacle data taken from real vehicle runs using Smarty in conjunction with a laser range finder. For the path arbiter, the FOLLOW PATH behavior issued a line of positive utility, along with intermediate points, originating from the vehicle start point at an angle of 20 degrees, and for the turn arbiter the FOLLOW HEADING behavior continually issued arc votes based on servoing to that line. To keep the experiment as simple as possible and reduce confounding variables, the MAINTAIN TURN and DRIVE STRAIGHT behaviors were not used with the turn arbiter, and the corresponding biases in the path arbiter were set to zero. For all test runs, the vehicle was placed at the start pose of x = -20 meters, y = -50 meters, and yaw = -90 degrees (facing in the direction of the positive X-axis). The run was terminated as soon as the vehicle crossed the “finish line” at y = -35 meters.

Experiments were done in simulation varying four variables:

- Arbiter: Turn Arbiter (no prediction capability), Path Arbiter (predictive control off), or Path Arbiter (predictive control on)
- Speed: 1 through 6 meters/second, in 1 meter/second increments
- System Latency: 0, 1, or 2 seconds
- Actuator Response Time: curvature could either change instantaneously or at the rate of 0.14 radians/seconds^2.

The dependent variables being measured are:

- Path Smoothness, as defined in Equation 5-3
- Mean Obstacle Proximity, as defined in Equation 5-1

Because the driving factor in these experiments was obstacle avoidance, mean goal distance did not provide a meaningful measure of performance, and so was not used.
5.5.2 Experimental Results

System Latency = 0, Actuator Acceleration = $\infty$

The vehicle paths under this condition all were essentially the same as the one shown in Figure 5-13, independent of vehicle speed or type of arbitration.
As can be seen in Figure 5-14, the mean obstacle proximity remains very close to zero at all speeds and for all arbiter types, reflecting the fact that without the effect of dynamics, the arbiters always manage to keep the vehicle clear of all obstacles.

The curvature profile for these runs is shown in Figure 5-15. Since the vehicle is initially headed straight for one of the corridor walls, it first has to turn left to get aligned with the corridor, and then it just makes adjustments as it goes along until the wall on the left disappears and the vehicle turns left to get in the open, and then the right wall disappears and the vehicle makes a hard right to get back to its goal path. Because there are no delays or acceleration limits, the actual curvature and the commanded curvature coincide.

On the left column of Figure 5-15 are the profiles from runs at 1 meter/second, and on the right are runs at 6 meters/second; note that the time scale is compressed for the latter, resulting in greater curvature changes with respect to time, so that runs at higher speeds are jerkier as expected. Figure 5-16 shows that smoothness, which measures the first derivative of curvature with respect to time, rises slightly with speed for the path arbiter with or without predictive control, but more so for the turn arbiter. The graphs for smoothness of commanded and actual vehicle curvature are the same for this case.
Figure 5-15: Curvature profiles for turn and path arbiters at varying speeds.

a) turn arbiter at 1 m/s, b) turn arbiter at 6m/s

(c) path arbiter w/o predictive at 1 m/s, d) path arbiter w/o predictive at 6m/s,
e) path arbiter with predictive at 1 m/s, f) path arbiter with predictive at 6m/s
The introduction of finite actuator responses has the effect of introducing a low pass filter on the commanded curvature. While the commanded curvature is the same as the previous case without system latencies, the actual vehicle curvature is damped, as can be seen in Figure 5-17. However, without the introduction of system latency, the vehicle is still able to negotiate the corridor without approaching the obstacles on either side, so the mean obstacle proximity for all runs under this condition remains nearly zero as with the previous case.

The introduction of a 1 second delay between the time a turn command is issued and the time the vehicle begins to execute that turn results in the expected discrepancy in commanded and actual
curvature show in Figure 5-18. Because the two plots are identical except for a phase shift, the smoothness metric is unaffected and remains like that of the case with no vehicle dynamics shown in Figure 5-16.

![Figure 5-18: Commanded vs. actual curvature with system latency of 1 second.](image)

Under these conditions, the path arbiter was still able to function as before, but the turn arbiter did not fare very well at higher speeds, as is made evident by the vehicle paths shown in Figure a; Figure b shows that the resulting mean obstacle proximity metric for those runs is very high while all others are still nearly zero.

![Figure 5-19: Degradation of turn arbiter performance with 1 second system latency.](image)

a) Vehicle paths executed by turn arbiter at speeds of 5 and 6 meters per second, b) mean obstacle proximity as a function of speed for path and turn arbiters
This case is almost identical to the one that precedes it where the system latency was also 1 second but there was no limit on actuator acceleration. Like that case, the vehicle successfully negotiated the corridor and thus the mean obstacle proximity was low, except for when the turn arbiter was attempting to control the vehicle at higher speeds; the resulting paths for the latter condition is shown in Figure a. Note that the paths are much rounder than in the previous case; this is the one effect of modeling finite actuator response. While the commanded turns are the same as before, we once again get the effect of a low pass filter, producing smoother paths as can be seen in Figure b. Although the paths are smoother, this merely reflects the fact that the turn arbiter has less control than before, and should not be taken to indicate improved performance.

**Figure 5-20**: Degradation of turn arbiter with 1 second latency and actuator limit.

- a) Vehicle paths executed by turn arbiter at high speeds,
- b) commanded vs. actual curvature

In this case the system latency is increased to the value of 2 seconds, which is not unrealistic in a distributed system communicating via a medium that has no upper bound on delays and processing on an operating system that is not real-time. The graphs of mean obstacle proximity as a function of speed in Figure 5-21a and of path smoothness vs. speed in Figure 5-21b show that the turn arbiter does very badly once again at higher speeds; these runs are shown in Figure 5-22. The graphs also show that, at higher speeds, the path arbiter without predictive control performed even worse than the turn arbiter, possibly due to the path arbiter’s greater complexity. The turn arbiter runs at speeds of 5 and 6 m/s are shown in Figure 5-23. However, when the path arbiter made use of its predictive control capabilities, it was still able to go through this narrow corridor at reach the goal, in spite of the fact that a delay of 2 seconds at a speed of 6 meters/second meant that the vehicle would have travelled 12 meters between the time that a command was issued and
the time that it would actually be executed. These successful path traces are shown in Figure 5-24, along with the trace of the position of the vehicle as predicted by the arbiter, which for the most part coincides well with the actual path.

Figure 5-21 : Path metrics as a function of speed.
a) mean obstacle proximity and b) path smoothness.

Figure 5-22 : Turn arbiter: paths executed at high speeds with 2 second latency.
Figure 5-23: Path arbiter w/o prediction: vehicle paths at high speeds.

Figure 5-24: Path arbiter with prediction: actual and predicted paths at high speeds.
5.5.3 Summary of Experimental Results

Effects of Latency on Turn Arbiter

Mean Obstacle Proximity

Mean obstacle proximity was low under most conditions, but as expected rose significantly when significant vehicle dynamics were introduced and the vehicle was moving at a higher speed, as shown in Figure 5-25.

![Figure 5-25](image_url)  
**Figure 5-25:** Turn arbiter: effect of system latency on mean obstacle proximity.

Path Smoothness

Again as expected, smoothness got worse at higher speeds, but only when the simulated controller was capable of infinite accelerations, as can be seen in Figure 5-26. As discussed previously, the finite actuator response acted as a low pass filter, yielding runs that are smoother but not necessarily better in terms of quality of control, which is better indicated by the mean obstacle proximity.
Effects of Latency on Path Arbiter without Predictive Control

**Mean Obstacle Proximity**

For almost all conditions, mean obstacle proximity was much lower for the path arbiter than the turn arbiter, even without the use of predictive control. As can be seen in Figure 5-27, the one exception when performance was worse was when the system latency was 2 seconds, the actuator response delay was on, and the vehicle speed was at least 5 meters per second. Perhaps the greater complexity of the path arbiter compared to the turn arbiter resulted in decreased throughput in the
presence of so many obstacles to process, and in these worst case conditions the added latency was enough to result in degraded performance.

Figure 5-27: Path arbiter w/o prediction: effect of latency on obstacle proximity.

Path Smoothness

Again the actual path smoothness measure was low for all conditions, as shown in Figure 5-28a. Furthermore, closer examination in Figure 5-28b reveals that the smoothness measure was highest (i.e. the path was less smooth) for the case where no vehicle dynamics were simulated. This is because the simulator was able to respond to the commands as issued to carefully navigate through the path, while the introduction of vehicle dynamics effectively places a low pass filter on the turn commands issued. If the smoothness of the commanded vehicle curvature is looked at instead as in Figure 5-29, then we see that the measure is much larger in the case with the most significant dynamics, as was expected.
Figure 5-28: Path arbiter w/o prediction: effect of latency on actual path smoothness. a) full range, b) zoomed in.
In the case of the path arbiter using predictive control, execution was fairly flawless in almost all conditions, resulting in extremely low mean obstacle proximity values, as shown in Figure 5-30. Ironically, the exception was when the vehicle system had no delays modeled, in which case the predictive control was counterproductive because the model used did not match the conditions present.
As with the mean obstacle proximity, the path smoothness was highest for the case where no vehicle dynamics were simulated, due to both the inaccurate model and the lack of vehicle dynamics acting as a low pass filter. This is shown in Figure 5-31.

**Figure 5-30**: Path arbiter with prediction: effect of latency on mean obstacle proximity.

**Path Smoothness**

As with the mean obstacle proximity, the path smoothness was highest for the case where no vehicle dynamics were simulated, due to both the inaccurate model and the lack of vehicle dynamics acting as a low pass filter. This is shown in Figure 5-31.
5.6 Conclusion

Experiments were conducted using the turn and path arbiters on the vehicle in the slag heap test area. The arbiters received input from one behavior that processed range data to detect and avoid obstacles and another that directed the vehicle towards user-specified goals. For these experiments at slow speeds, both arbiters were able to successfully achieve their mission. In order to compare the performance of these arbiters under more carefully controlled conditions and at higher speeds, a simulation testbed was developed and the arbiters were run under a variety of conditions. The effects of system latency and dynamics became very apparent at higher vehicle speeds, and the path arbiter with predictive control performed much better than the turn arbiter under those conditions.

Figure 5-31: Path arbiter with prediction: effect of latency on smoothness.
6.1 Architectures for Mobile Robot Control

Mobile robots provide a useful and challenging domain in which to develop the thesis put forward here. Effective control of mobile robots and their associated sensors demands the synthesis and satisfaction of several complex constraints and objectives in real-time, particularly in unstructured, unknown, or dynamic environments such as those typically encountered by outdoor mobile robots. An architecture must connect the perception, planning, and control modules and provide a structure with which the system may be developed, tested, debugged, integrated and understood.

In addition to serving as an interesting case study for the development of control architectures, mobile robots have many uses, especially for gaining access to areas that are unreachable by or dangerous to humans such as cleanup of hazardous waste sites, inspection of nuclear power stations, military reconnaissance missions, subsea exploration, and planetary exploration. Other uses include the automation of routine tasks such as mail delivery and the inspection and waterproofing of heat tiles underneath the space shuttles. Recently, a new emphasis has been the incorporation of robotic technology in passenger vehicles in order to improve both the safety and efficiency of highway driving.

In order to function in unstructured, unknown, or dynamic environments, a mobile robot must be able to perceive its surroundings and generate actions that are appropriate for that environment and for the goals of the robotic system. One of the key characteristics of most mobile robot domains is that uncertainty plays a large role, and that is the class of domains studied in this work. Prior knowledge of the environment may be incomplete or not exist at all, and the environment may be dynamic, so that reasoning must be non-monotonic in nature and occur rapidly enough to be able to respond to unexpected events. Posterior knowledge gained via sensing is incomplete, inaccurate, and uncertain, as is knowledge of the effects of actions decided upon and taken by the system. In addition, the dynamics of the vehicle itself often play an important role in determining which actions may be achieved and which actions are to be avoided.

Another important aspect of mobile robot systems is the need to combine information from several different sources such as video cameras, laser range finders, sonars, and inertial navigation systems; they must also be capable of combining objectives for a system that is to perform diverse tasks such as following roads, driving off-road, following designated paths, avoiding obstacles, and reaching goal destinations, as well as allowing for teleoperation. These sensors and the procedures that process their data operate at different rates, so they must be allowed to operate asynchronously to maximize the throughput and thus the responsiveness of the system. To function effectively, an architectural framework for these sensing and reasoning processes must be imposed to provide a structure for combining information from several different sources. The architecture should serve as an aid, not a burden, in the integration of modules that have been developed independently, so it must not be overly restrictive. It must
allow for purposeful goal-oriented behavior yet retain the ability to respond to potentially dangerous situations in real-time while maintaining enough speed to be useful. While many general purpose architectures are computationally equivalent or nearly so, the ease with which various classes of systems may be instantiated within them varies greatly. A crucial yet often neglected consideration in the design of a mobile robot architecture is the ease with which a system may be developed, tested, debugged, and understood, as well as adapted to novel domains and applications.

When dealing with a physical system such as a mobile robot, it is also important to consider aspects of control such as stability and the limitations and constraints of the physical plant, such as the non-holonomic constraints of a wheeled vehicle, finite actuator capabilities, and the delays inherent in any system that arise from latencies in data acquisition, data processing, intermodule communications, and actuator response; together with the continuous motion of the vehicle, this implies that by the time the command is being executed the vehicle is no longer in the current state but actually in a future state, i.e. different position, heading velocity, turn rate, etc. An asynchronous distributed system presents an additional challenge in that, in general, the size of these latencies will be different for each module due to varying processing needs and sensor frame rates. If the various latencies of the system are not accounted for, the vehicle control will be unstable.

When designing a software architecture for the control of complex real-time systems such as mobile robots, there are many important issues that must be addressed. The architecture must provide the means by which the system may accomplish its multiple objectives efficiently; it must be able to satisfy real-time constraints, promote fault tolerance, and provide for the safety of the vehicle and of its surroundings. Some key issues to be considered in the design of a mobile robot control architecture are whether the architecture should be centralized or distributed, whether the reasoning should be reactive or deliberative, whether input combination should occur via sensor fusion or command arbitration, and whether control should be top-down or bottom-up. To be effective in dynamic, uncertain, and complex domains, DAMN combines both deliberative and reactive elements within a bottom-up structure, and centralized arbiters combine commands received from distributed, asynchronous behaviors.

Deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other’s deficiencies. Reactive components provide the basic capabilities which enable the robot to achieve low-level tasks without injury to itself or its environment, while deliberative components provide the ability to achieve higher-level goals and to avoid mistakes which could lead to inefficiencies or mission failure. Because reactivity is essential for any real-time system, we must eschew the sensing and planning bottlenecks of centralized systems, but if we are to avoid sensor fusion, the system must combine command inputs to determine an appropriate course of action. Having decided upon a behavior-based architecture, it is then necessary to specify how information in a distributed system is to be combined and used to generate action in an intelligent, coherent manner. By appropriately combining behavior commands through arbitration, a robot
Chapter 6: Conclusion

control system can respond to its environment without suffering the problems inherent in sensor fusion. While sensor fusion creates a bottleneck, command arbitration runs the risk of losing information valuable to the decision-making process; therefore a careful balance must be struck between completeness and optimality on the one hand versus modularity and efficiency on the other. Systems that combine command inputs to determine an appropriate course of action fall into two broad categories, priority-based arbitration and command fusion.

In architectures which employ priority-based arbitration such as the Subsumption Architecture, action selection is achieved by assigning priorities to each behavior; of all the behaviors issuing commands, the one with the highest priority is in control and the rest are ignored. This allows for quick responses to new situations and stimuli, although, by definition, prioritization only allows one module to affect control at any given time. While this is an effective scheme for choosing among incompatible commands, it does not provide an adequate means for dealing with multiple goals that can and should be satisfied simultaneously. A compromise between behaviors cannot be achieved in such an all-or-nothing scenario; whenever one behavior’s output is overridden by another, the information and knowledge represented by that behavior is completely lost to the system.

Architectures that perform command fusion combine the commands from individual behaviors so that decisions may be made based on multiple considerations while preserving the modularity and reactivity of distributed systems. Command fusion provides a mechanism for the concurrent satisfaction of multiple goals, and allows modules to be completely independent, thus allowing incremental, evolutionary system development. In systems that perform command fusion, the decision-making process is based on combining command inputs from multiple behaviors, and is therefore able to simultaneously satisfy multiple goals. Such systems are largely distributed, but they must also contain a central module that receives commands from the various behaviors and combines them in some manner. Motor schemas provide a general framework for command fusion, but because they are implemented as potential fields they are subject to the problems of command averaging and local minima, among others.

6.2 Contributions

6.2.1 Novel Paradigm for Command and Utility Fusion

The Distributed Architecture for Mobile Navigation (DAMN) is a planning and control architecture in which a collection of independently operating behaviors collectively determine a robot’s actions. The organization of the DAMN architecture is shown in Figure 6-1; it consists of a group of distributed behaviors communicating with a centralized command arbiter, sending votes in favor of actions that satisfy its objectives and against those actions which do not. The arbiter is then responsible for combining the behaviors’ votes and generating actions which reflects their objectives and priorities. Within the framework of DAMN, behaviors must be defined to provide the task-specific knowledge for controlling the vehicle. Each behavior in the system is responsible for a particular aspect of vehicle control or for achieving some particular
task, receiving as input only that sensory information pertinent to the task, and processing that input to generate as output votes for and against possible actions. Each behavior operates asynchronously and in parallel with other behaviors, sending its outputs to the arbiter at whatever rate is appropriate for that particular function. A behavior is assigned a weight reflecting its relative priority in controlling the vehicle; however, this weighting is not a strict prioritization, thus votes from all behaviors are used in determining what the next action should be. A mode manager may also be used to vary these weights during the course of a mission based on knowledge of which behaviors would be most relevant and reliable in a given situation. These behavior votes are then periodically combined by the arbiter and the resulting command is sent to the controller.

![Diagram](image)

**Figure 6-1**: Centralized arbitration of votes from distributed behaviors in DAMN.

The DAMN architecture reflects the position that some centralization is needed to provide coherent rational reasoning, but an appropriate level must be chosen so that it does not create a bottleneck. The capabilities of an agent should be divided up as finely as is practical among task-achieving behaviors, and the interfaces must be defined so as to avoid being overly restrictive. Thus, as shown in Figure 6-1, independent behaviors operate in a distributed fashion to generate votes for actions based on domain-specific knowledge, while a central arbiter combines their results to generate reasonable behavior that satisfies as many objectives as possible. Thus, DAMN performs centralized arbitration of votes from distributed, independent, asynchronous decision-making processes and in so doing provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment.

Likewise, deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other’s deficiencies. Reactive components provide the basic capabilities which enable the robot to achieve low-level tasks without injury to itself or its environment, while deliberative components provide the ability to achieve higher-level goals and to avoid mistakes which could lead to
inefficiencies or mission failure. Unlike other behavior-based architectures, DAMN is designed so that behaviors provide both deliberative and reflexive capabilities; the use of distributed shared control allows multiple levels of planning to be used in decision-making without the need for an hierarchical structure. Input from all behaviors are used in determining what the next action should be, so that there are multiple influences on the decision-making process, and compromises are made to satisfy as many objectives as possible. In order to achieve this, a common interface is established so that modules can communicate their intentions without regard for the level of planning involved. The distinction made in DAMN is not in the level of abstraction of a given module, but rather whether its domain is represented and acted upon in a discrete or continuous manner. All continuous servo-like activity is instantiated as a voting behavior without regard for the time or space scale in which it operates; votes from behaviors operating at all levels of reasoning cooperate in determining what the next action should be. Sequential activity is controlled by a mode manager which exerts meta-level control within DAMN by modifying the voting weights assigned to behaviors and thus controlling the degree to which each behavior may influence the system’s decision-making process and thus the robot’s actions.

In addition, the behaviors are completely modular and independent, so that new capabilities may be added to an existent system without a need to disrupt or modify previously established functionality; in particular, they operate asynchronously, each at their rate without the need to synchronize with a central clock, so that throughput and reactivity is maximized. Thus DAMN provides a framework for facilitates the evolutionary development and integration of independent decision-making modules to create robust systems of incrementally greater capabilities.

**Command Fusion**

DAMN was devised to provide a general framework for command fusion that overcomes the limitations of priority-based arbitration such as that used in the Subsumption Architecture, as well as other command fusion systems such as Motor Schemas.

DAMN command arbiters combine votes for and against each of a set of candidate commands, either directly in actuator space or indirectly in effect space. This method has the benefit of being very simple and straightforward, so that many different types of modules may easily be incorporated within the architecture. Voting takes places in a commonly defined action or state space, thus providing a uniform interface to the arbiter that is independent of the task the behavior is concerned with, the level of planning at which it operates, or the algorithms and paradigms which it uses for its decision-making process. DAMN is designed so that various behaviors can be easily added or removed from the system, depending on the current task at hand. The distributed, asynchronous nature of the architecture allows multiple goals and constraints to be fulfilled simultaneously, thus providing goal-oriented behavior without sacrificing real-time responsiveness. Unlike other behavior-based architectures, DAMN is designed so that behaviors provide both deliberative and reflexive capabilities; the use of distributed shared control allows multiple levels of planning to be used in decision-making without the need for an hierarchical structure.
Within DAMN, independent behaviors operate in a distributed fashion to generate votes for actions based on domain-specific knowledge, while a central arbiter combines their results to generate reasonable behavior which obeys all constraints and simultaneously satisfies as many objectives as possible by choosing that action which maximizes a function of the behavior votes. In order to preserve the respective advantages of centralized and distributed architectures, sufficient information must be communicated from the behaviors to allow the arbiter to make intelligent decisions, but the arbiter must not be so complex as to become a bottleneck for the system. Various points along this trade-off spectrum have been explored within DAMN, using different types of arbiters and vote structures. There are currently four different arbitration schemes in effect, illustrated in Figure 6-2.

The simplest DAMN arbitration scheme is constraint arbitration, in which each behavior determines the highest value of the command being determined that satisfies the constraint which it seeks to enforce; the behaviors send that maximum value to the arbiter, which then takes the minimum of these values, thereby simultaneously satisfying all constraints. For example, speed behaviors determine their maximum speed vote based on constraints such as avoiding vehicle tip-over and avoiding obstacles, and the arbiter goes at the speed of the slowest vote, as shown in Figure 6-2a.

In the actuation arbitration scheme, each behavior votes for or against various alternatives in the actuator controller’s command space; for example, the turn arbiter receives votes for a fixed set of vehicle turn radii, as shown in Figure 6-2b. The arbiter then sums these votes and selects that action which has the highest total score, and send the command to the controller.
In the effect arbitration scheme, each behavior votes for or against various alternatives in an abstract command space, i.e., they vote for the desired effect of the mechanism being controlled rather than the direct control of the mechanism’s actuators. For example, a field of regard arbiter and its associated behaviors have been implemented and used for the control of a pair of stereo cameras on a pan/tilt platform, as shown in Figure 6-2c.

Utility Fusion

A new means of action selection via utility fusion is introduced as a solution to the shortcomings of behavior-based systems. Instead of voting for actions, behaviors indicate the utility of various possible world states, and it is the responsibility of the arbiter to determine which states are actually attainable and how to go about achieving them. This type of arbiter is no longer performing command fusion, nor is it performing sensor fusion; it is combining utilities to perform utility fusion.

This new approach strikes a balance between the extremes of action selection and sensor fusion and has been found to yield many benefits. The arbiter determines the next action based on the maximization of expected utility, thus providing a unified conceptual framework for defining the semantics of votes and for dealing with uncertainty. By explicitly representing and reasoning about uncertainty within the decision-making processes, a system can be created whose effects are well-defined and well-behaved. For example, a map-based path arbiter has been implemented as a means of voting for and producing steering control. Behaviors communicating with the path arbiter vote on the desirability of various possible vehicle locations, and the arbiter maintains a local map of these votes, as indicated in Figure 6-2d. The map-based utility space is not time-dependent, so that an arbiter using such a representation is capable of effectively synchronizing and maintaining a consistent interpretation of the votes received from asynchronous behaviors, thus providing coherent reasoning in a distributed system. The path arbiter evaluates the candidate trajectories within the utility map, and selects that one for which the total utility is the greatest. Behaviors can function without knowledge of the system dynamics, thus increasing their reusability for other systems.

The utility arbiter can use models of the system being controlled to determine which states are actually attainable, and to increase the accuracy and stability of control. In particular, the map-based utility arbiter gathers information from behaviors about the desirability of possible vehicle locations and then evaluates candidate trajectories to determine appropriate actions. The arbiter can then use kinematic models of the robot to determine which actions can be commanded without violating non-holonomic constraints, and use and dynamic models of the system to provide greater stability. Behaviors can function without knowledge of the system dynamics, thus increasing their reusability for other systems and types of vehicles.

The utility arbitration system does not perform sensor fusion, but instead the behaviors process data according to their tasks in a distributed fashion. However, the arbiter is more complex than those performing command fusion and resulting the system is thus more centralized than
command fusion systems. As a result, systems implemented with utility fusion are less reactive but are better able to anticipate circumstances and to generate coherent action. In DAMN, the processing of sensory information is still distributed among the behaviors, and the utility fusion performed by the arbiter is much simpler and represents much less of a bottleneck than does sensor fusion. By centralizing the action evaluation process, the type of DAMN arbiter is able to address issues discussed such as synchronization of commands and stability of control. Utility theory also provides a semantic framework to define the meanings of votes, and to separate out the effects of uncertainty and reason about them explicitly.

Thus, through the appropriate use of command or utility fusion, DAMN performs centralized arbitration of votes from distributed, independent, asynchronous decision-making processes and in so doing provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment. DAMN also provides a framework for developing and integrating independent decision-making modules communicating with such arbiters, thus facilitating their development and leading to evolutionary creation of robust systems of incrementally greater capabilities.

### 6.2.2 Integration of Heterogenous Modules

My work in building and testing DAMN has created a flexible and useful robot architecture that contributed significantly to the successes of the CMU Navlab project, the DARPA Autonomous Land Vehicle program, and the ARPA/OSD Demo II program. Vehicles under the control of DAMN have driven at highway speeds, navigated across stretches of off-road terrain some kilometers in length, cooperated with other robotic vehicles, and performed teleoperation, all while providing for the safety of the vehicle and meeting mission objectives; the types of data used in these systems include maps, color cameras, stereo cameras, sonars, ladar, and motion and position sensors.

DAMN has been used to combine various systems of differing capabilities on several mobile robots, at various sites; in addition to its use on the CMU Navlab vehicles, DAMN is also being used at the Lockheed Martin Corporation, the Hughes Research Labs, and the Georgia Institute of Technology. DAMN arbiters have been used to integrate navigation modules for the steering and speed control of single as well as multiple vehicles at these sites, and have also been used to select field of regard for the control of a pair of stereo cameras on a pan/tilt platform.

By providing a behavior general interface at the geometric level; the DAMN architecture has been able to integrate a wide variety of existing modules with minimal modification, regardless of those modules’ internal representations, behavioral semantics, or the level of planning at which they operate. Thus, it provides new functionality to a system by synergistically combining existing capabilities. It does so without forcing the modules to be developed or reformulated to fit within a preconceived notion of how they should function.
6.3 Future Work

6.3.1 Definition of Uncertainties

As currently defined, the utility arbiter does not deal with uncertainty in an entirely satisfactory manner. As a behavior informs the arbiter of new utilities, they either replace old ones from that behavior or are included in the map together without any sort of combination taking place. When state space trajectories are evaluated, their expected utilities are summed. While this is correct for independent utilities corresponding to distinct objects, uncertainties for the same object should be combined via Bayes’ theorem. However, this implies that the arbiter must know whether or not observations are independent, both from the same behavior and from different behaviors, and the complexity of the system would cause a significant bottleneck comparable to that created by sensor fusion methods.

Utility theory does provide a framework in which the uncertainties due to sensing, position and control can be expressed explicitly and reasoned about in a rational manner. However, the current system as implemented does not make full use of this capability. The uncertainty in sensing is specified as an arbitrary constant for a given behavior, rather than using measurements of the uncertainty in the sensor, which may also vary within an image, and of the sensor processing algorithm, which may vary depending upon the characteristics of the image.

Uncertainty in position and control should be used in the decision-making process as well. Currently trajectories are evaluated from a single predicted vehicle position rather than a distribution of possible locations due to uncertainty in position and control, as well as in the prediction process. Expected utility is defined by summing, for each consequence, the product of the utility of that consequence and the probability that it will occur. Thus, uncertainties can be convolved to calculate the probability of the various consequences which have been assigned utilities, and an integral performed to measure the true expected utility of an action.

6.3.2 Definition of Utilities

While it was hoped that utility theory would provide an objective means of combining information from disparate sources, utilities are still defined in a subjective manner, according to intuition and experience, just as weights are defined in a command fusion system. This appears to be a fundamental problem of multiple-objective systems, whether distributed or centralized, which is not completely solved by utility theory. However, user protocol studies could be conducted so that the utilities, while still subjective, may be defined within a rational framework in accordance with a model of the human decision-making process. Machine learning techniques such as reinforcement learning or genetic algorithms could also be employed in order to learn utilities and weights through experience, and to adapt parameters to the current situation.

In addition, the utilities as they are currently defined are valid from the particular pose from which the behavior performed the evaluation. For example, the utility expressed for a particular location
specified in x-y space may be based on the assumption that the vehicle would go directly to that location from its current position; it is not necessarily the case that the same utility would be assigned if the vehicle were to approach it later from a different point with a different orientation. Other dimensions such as orientation could also be added to the representation, yielding a planner capable of planning in configuration space.

### 6.3.3 Time-Dependent Planning

Time is another dimension not explicitly represented by the map-based utility arbiter. Although the arbiter is able to explicitly reason about the movement of the vehicle, the utilities in its map are static so that it is necessary for behaviors to quickly update the location of a utility, even when the object to which it corresponds is moving with a known trajectory. In order to perform a limited form of time-dependent planning, the representation for utilities could be extended to include velocities when known, and the arbiter would then update the position of these utilities as time progresses, as well as in anticipation of system latencies as was done for predictive control. When evaluating a trajectory, the arbiter would also have to account for the time taken for the vehicle to move along the path, updating the predicted position of utilities as the position being evaluated progresses.

### 6.3.4 Coordination of Multiple Degrees of Freedom

There is usually a one-to-one mapping between an arbiter and a control degree of freedom such as steering or speed, although they are in fact interdependent and should be jointly controlled, as was the case with pan/tilt camera control. This is a challenge for higher degree of freedom systems such as robot manipulators or multiple coordinating robots. The coordination of multiple degrees of freedom may be accomplished by creating a multi-dimensional command space and requiring that behaviors vote for all combinations of their cross-product, as shown in Figure 6-3. However, the combinatorics require prohibitively expensive computations on the part of all behaviors, as well as the arbiter; this is exacerbated as the dimensionality or the resolution within any single dimension increases.

![Figure 6-3: Two-dimensional coordinated speed and turn arbitration.](image)

The map-based utility arbitration scheme provides yet another possible means of coordinating multiple degrees of freedom. The trajectories evaluated depend upon the current vehicle speed. If instead the arbiter were to evaluate trajectories at several different speeds, it could then choose that turn and speed combination which maximizes expected utility overall. While this induces
some combinatorics within the arbiter, it can be partially compensated for by precomputing the
trajectories at the pre-determined speeds. One significant advantage of this scheme, however, is
that behaviors incur no greater structural or computational overhead; as before, they simply
express the utility of various external world states, and the arbiter exercises more control over
which states may be achieved and how. This means of jointly determining turn and speed
commands has not yet been implemented at this time.

It is also possible to place arbiters in nested control loops, with a map-based utility arbiter
providing inputs to an effect arbiter which in turn feeds an actuation arbiter whose output is used
by a constraint arbiter. For example, the path arbiter and turn arbiter could be nested as shown
schematically in Figure 6-4. The path arbiter performs evidence fusion and evaluates candidate
actions, sending the results as votes to the turn arbiter which performs command fusion and
selects an action to be executed, and sends that turn command to the path arbiter as feedback.

![Figure 6-4: Multiple arbiter nested control loop.](image)

Path arbiter sends evaluations as votes to turn arbiter and receives turn commands as feedback.

### 6.4 Conclusion

Because reactivity is essential for any real-time system, we must eschew the sensing and planning
bottlenecks of centralized systems, but if we are to avoid sensor fusion, the system must combine
command inputs to determine an appropriate course of action. However, priority-based arbitration
only allows one module to affect control at any given time. Command fusion provides a
mechanism for the concurrent satisfaction of multiple goals, and allows modules to be completely
independent, thus allowing incremental, evolutionary system development.

The Distributed Architecture for Mobile Navigation is a planning and control architecture in
which a collection of independently operating behaviors collectively determine a robot’s actions.
A command arbiter combines the behavior outputs and selects that action which best satisfies the prioritized goals of the system. The distributed, asynchronous nature of the architecture allows multiple goals and constraints to be fulfilled simultaneously, thus providing goal-oriented behavior without sacrificing real-time responsiveness. Unlike other behavior-based architectures, DAMN is designed so that behaviors provide both deliberative and reflexive capabilities; the use of distributed shared control allows multiple levels of planning to be used in decision-making without the need for an hierarchical structure.

Thus, DAMN performs centralized arbitration of votes from distributed, independent, asynchronous decision-making processes and in so doing provides coherent, rational, goal-directed behavior while preserving real-time responsiveness to its immediate physical environment. DAMN also provides a framework for developing and integrating independent decision-making modules communicating with such arbiters, thus facilitating their development and leading to evolutionary creation of robust systems of incrementally greater capabilities.

DAMN has been used to combine various systems of differing capabilities on several mobile robots, and has also been used for active sensor control. Various subsystems developed at CMU and elsewhere have been integrated within this architecture, creating systems that perform road following, cross-country navigation, map-based route following, and teleoperation while avoiding obstacles and meeting mission objectives.
Appendix A: Motor Schemas

Motor Schema theory has been developed in order to provide a biologically plausible model of the behavior of various animals in certain complex situations [3]; it has also proven to be an useful and interesting framework for applying Distributed Artificial Intelligence techniques to the robotics domain. It consists of perceptual schemas which map sensed data into object representations which in turn are used by motor schema to generate actions. Motor schema are similar to behaviors in many respects; however, rather than using arbitration to select the output of a single behavior, the outputs of all schema are combined in order to produce an action that reflects multiple considerations. This combination of schema is also prioritized, in that each schema has a weight associated with its output command that reflects the importance of the command or the certainty of the information on which it is based.

Arbib and House developed two different schema models which could explain prey-catching and obstacle-detouring behaviors that were observed in frogs [5]. The Vector Field model of path planning associates motor actions with schema objects and combines them; the Orientation Selection model chooses among orientations based on a summation of excitatory and inhibitory influences from each schema. Arbib has pursued the Vector Field model in his subsequent work [4], and this approach has become the robot control paradigm associated with motor schema theory [6]; however, it is not the only possible instantiation of the theory. Although they were not explicitly developed for the purpose, there exist robot control systems which serve as realizations of the Orientation Selection model. One such system is the Vector Field Histogram [11], another is the Distributed Architecture for Mobile Navigation.

A comparison of the two schema paradigms will be made in the context of explaining observed animal behavior, and instantiations of these paradigms will be compared in the context of controlling a mobile robot. In order to understand motor schemas, let us first study the domain for which they were first developed: the observed prey-catching and obstacle-avoiding behavior of frogs.

A.1 Observed Prey-catching Behavior of Frogs

When frogs are presented with stimuli such as worms, the animal selects one of the prey and moves directly towards it, as shown in Figure A-1a; this can be modeled in a classical stimulus-response fashion with each potential prey triggering an excitatory response in the frog. When obstacles such as fence posts are introduced, they influence the decision of which direction to head in, as shown in Figure A-1b; the frog circumvents the fence and then heads toward the worms. However, a simple stimulus-response mechanism cannot explain the behavior observed in Figure A-1c, where the frog moves around the fence posts only to find itself blocked by a solid wall; the frog continues to orient itself toward the chosen prey even though it is no longer visible.
The selection of movement direction as described above is essentially one-dimensional; it depended only on the angles between the frog and its prey and between the frog and the obstacles. However, it is clear from further experimentation that depth information also plays a significant role in the decision-making process. Figure A-2 shows how the behavior of the frog varies under different conditions; the arrows indicate the directions along which the frog moved, along with the percentage of trials for which each direction was chosen. Figure A-2a shows that the frog preferred to go around the fence posts to get to the worm, while Figure A-2b shows that a gap in the fence was taken advantage of when present. In Figure A-2c, the closer fence has a gap in it and the further one does not; the frog chose to go through the gap in the first fence more often than it chose to go around it, yet it did not go through the gap so often as if the second fence were not there. Neither fence in Figure A-2d contained a gap, and so the frog almost always chose to detour around them. In Figure A-2e, a large gap in the closer fence led the frog to often choose to go through rather than detour around, even though the side barriers formed a trap. These observed behaviors indicate that both fences contributed to the decisions made, and that the influence of perceptual information diminishes with the perceived object’s distance; the frog is apparently maintaining and using depth information, but it is not processing it in a manner sophisticated enough to account for the more complex topology of the last case. Computational models of the frog’s behavior must account for these observed phenomena, as well as the fact that, even under the same initial conditions, the same decision to go through versus detour around the fence is not always made.

Figure A-2: Frequency with which frog went through or around a fence.
A.1.1 Schema Models

Schema theory has been used to create models which account for the experimental results described above in a way that is cognitively, neurologically and computationally plausible. Schemas are independent modules which represent perceptual information and contain processes for deciding how to act on that information; together, they constitute a distributed system for reasoning and for acting, either externally to change the state of the physical world or internally to change the state of the system. Each schema has an associated activity level which represents its confidence that the perceptual information it contains is accurate.

In applying schema theory to the domain described above, Arbib and House postulate that two separate depth maps are maintained for prey stimuli and obstacle stimuli, that they are individually processed to generate orientation preferences, and that the motor schemas combine these preferences in order to produce coordinated action [5]. The methodology outlined by schema theory is broad enough that it may be implemented by more than one computational paradigm. Arbib and House postulated both an Orientation Selection model and a Vector Field model. Although only the latter approach was pursued in subsequent work in modelling animal behavior, either model could account for the experimental data on frogs which was presented, and each has their advantages and disadvantages. We present the two models described, and discuss the merits of each.

Orientation Selection Model

In the Orientation Selection model, the schemas operate by convolving input depth maps with masks designed to elicit the appropriate response in a two-dimensional distance/orientation space, as shown in Figure A-3a. The obstacle-avoidance schema convolves the barrier depth map with an inhibitory mask, shown in Figure A-3b, which has the effect of slightly spreading the obstacle laterally (in the orientation dimension); the mask also spreads obstacles backward in distance so that potential goal locations behind the obstacle are also inhibited. The prey-catching schema convolves the prey depth map with an excitatory mask. As can be seen in Figure A-3c, this mask spreads the input stimulus forward along the distance axis so that it may overcome inhibitory influences directly in front of it, and also has a large lateral spread which serves to create secondary goal locations in those cases where the inhibition due to a barrier is not overcome. The resulting arrays output by these two schemas are then summed to create a net excitation array which represents the combination of all excitatory and inhibitory influences. This excitation array is integrated over distance for several orientations to yield an histogram of total excitation as a function of orientation; the orientation corresponding to the highest value in that histogram is selected as the direction in which the frog will move. A depth segmentation module determines the target depth for each orientation, presumably where the excitation reaches its maximum value along that direction; the resulting target depth map can then be indexed by the chosen orientation, so that the frog has now determined the angle in which it will head and the distance it will move in that direction; thus, an action is decided upon by combining the influences of all available inputs.
The schemas are combined by summing the vector fields. Figure A-5 shows the vector field resulting from the combination of schemas representing the frog facing a prey object behind a fence. Arbib and House hypothesize that a “tracking” creature such as a gerbil would then perform some sort of integration on these vector fields in order to plan a path, and that a field such as that in Figure A-5 would yield two “bundles” of trajectories, one leading around the fence to the left and the other to the right. In the case of a “ballistic” animal like the frog, however, they suggest that the vector field is instead processed to create a map of target locations.

**Figure A-4**: Vector fields.

a) attractive toward prey; b) repulsive from obstacle; c) leading around a series of obstacles; d) self-induced forward propulsion.

The schemas are combined by summing the vector fields. Figure A-5 shows the vector field resulting from the combination of schemas representing the frog facing a prey object behind a fence. Arbib and House hypothesize that a “tracking” creature such as a gerbil would then perform some sort of integration on these vector fields in order to plan a path, and that a field such as that in Figure A-5 would yield two “bundles” of trajectories, one leading around the fence to the left and the other to the right. In the case of a “ballistic” animal like the frog, however, they suggest that the vector field is instead processed to create a map of target locations.
A.1.2 Comparison of Behavior Models

Both the Orientation Selection (OS) model and the Vector Field (VF) model qualitatively reproduce the behavior of frogs observed in the scenarios shown in Figure A-2. The excitation array in the OS model and the convergence map in the VF model both contain peak values only at the prey location and at the ends of the row of obstacles, thus predicting, in agreement with experimental results, that the frog will either move directly through the fence towards the prey or to either end of the fence posts. Figure A-2 also shows the frequency in which the frogs chose to either go forward or detour; the computational models must also be able to explain this apparent non-determinism. Arbib and House were able to reproduce this phenomenon within the OS model by varying the lateral spread of excitation induced by the presence of a prey. A narrow spread diminished the total excitation present at the fence ends, so that the frog would choose to go forward through the fence, while a wider spread used on the same inputs resulted in the frog choosing the detour behavior instead. Variation of this spread parameter within the frog population or even within an individual frog due to motivational factors could explain their probabilistic behavior. No such parameter was identified for the VF model; however, if the schemas were assigned a weight which affected the vector field summation, then variation of these weights might produce the desired result. For example, a higher weight for the prey schema relative to the obstacle schema could result in the frog going directly through the fence, while a lower prey weight could result in the detour behavior.

Arbib and House contend that the VF model is more sophisticated than the OS model and better explains complex behavior, mainly due to the assertion that the OS model creates a “simple decision surface” while the VF model creates a “spatially encoded map of potential motor activity”. However, the convergence map in the VF model and the excitation array in the OS model are both two-dimensional surfaces whose peaks correspond to potential targets for the next movement. The OS model uses this map to create a one-dimensional histogram in order to select an orientation, but the VF model must ultimately process the convergence map in a similar manner in order to decide upon a specific target. Additionally, Arbib’s evaluation is based on the assumption that an entire path is planned before it is executed, rather than reevaluating the situation as the animal moves along. By continually deciding and choosing a new direction of movement based on the current situation, the OS model can produce arbitrarily complex paths. Arbib and House also claim that, in the OS model, once the frog has selected a target, it cannot remember whether that target corresponds to the location of a prey object or merely an intermediate destination in a detour maneuver; furthermore, if the frog is executing a detour behavior, it will not be able to remember where the original prey target was and therefore cannot orient towards it. However, the preprocessing stages in the orientation selection process generate and maintain essentially the same information as the vector fields; in particular, the input depth maps contain the location of both prey and obstacles, and the output of the depth segmentation module, combined with the orientation histogram, provides goal locations in polar coordinates. Thus, it appears that both the Orientation Selection and the Vector Field models could account for observed animal behavior, and that further experimentation is needed in order to distinguish the two as viable mechanisms. More importantly for our purposes here is the question of how
effectively these schema models may be applied to the robotics domain. Implemented physical systems can provide feedback regarding matters such as the suitability of the representations used, the efficiency of the algorithm, and the applicability to various other situations and environments.

**VECTOR FIELD HISTOGRAM**

The Vector Field Histogram [11] is a method used for robot control which serves as a realization of the Orientation Selection model, although it was not developed with that intention. The Vector Field Histogram (VFH) method begins by building an egocentric map of observed obstacles. It does so by creating a local histogram grid whose elements represent the number of times the robot’s sensors have reported the presence of an obstacle at that location, as illustrated in Figure a. This corresponds to the input obstacle depth map used in the OS model, with the addition of the histogram counts which provide the certainty information which Arbib hypothesized but did not include in his model. As in the OS model, the next stage of processing involves integration along various orientations to yield a one-dimensional polar histogram of obstacle density (Figure b); like the Vector Field model, the significance of a perceived obstacle diminishes with distance. Note that this histogram corresponds only to the inhibitory influence of obstacles in the OS model, rather than a general summation of excitation that includes other factors such as the presence of attractors. The histogram grid is not convolved with a mask as in the OS model, but the same effect of spreading activation is achieved in the VFH method by smoothing the polar histogram so that orientations adjacent to obstacle directions also receive inhibitory influence. Finally, a threshold $\tau$ is applied to the polar histogram, and those areas whose obstacle density is below that threshold are considered as candidate directions for movement. The presence of goals such as a prey object now comes into play, as that candidate direction which is closest to the direction of the goal is chosen as the direction of travel.

![Figure A-7: Vector Field Histogram.](image)

**POTENTIAL FIELDS**

Potential Fields is a method of planning robot trajectories based on combining the vector fields induced by mapped objects, such as repulsive fields from obstacles and attractive fields from goals [8] [37]. This approach to mobile robot control is very similar to the Vector Field model; as
problem, is that arbitration via vector addition can result in a command which is not satisfactory to any of the contributing behaviors. The DAMN system does not average commands, but rather selects the command which has the most votes from the behaviors, and the VFH system selects the direction that is closest to the goal direction and unobstructed by obstacles. A related problem with a potential field implementation is that the robot cannot pass through closely spaced obstacles. This effect, which can be observed in Figure A-4c, was desirable in modelling frog behavior because the frog did not pass through closely spaced fence posts unless there was a prey object immediately behind exerting a large attractive force; however, this is problematic for robots that wish to explore areas that may only be accessible by passing through narrow entrances such as doorways. Borenstein also reported conditions under which potential fields suffer from oscillations and instability [12]. Potential fields have the advantage that the entire path may be computed if the obstacle and goal locations are known \textit{a priori}; however, it has been found in practice that it is necessary for a robot to reevaluate its path often in order to maintain reactivity in a dynamic and uncertain environment. The most significant lesson learned from these implemented systems is that entire paths cannot be planned before any action is taken, and then blindly followed until the goal is achieved. Both animals and robots must be reactive to changes in their surroundings and to new perceptual information as it becomes available.
References


