A Portable Approach to High-Level Behavioral Programming for Complex Autonomous Robot Applications

Jesse G. Hurdus

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

Master of Science
In
Mechanical Engineering

Dr. Dennis Hong, Committee Co-Chairman
Assistant Professor of Mechanical Engineering
Virginia Tech

Dr. Alfred L. Wicks, Committee Co-Chairman
Associate Professor of Mechanical Engineering
Virginia Tech

Dr. Charles F. Reinholtz, Committee Member
Alumni Distinguished Professor of Mechanical Engineering
Virginia Tech
Chair of Mechanical Engineering
Embry-Riddle Aeronautical University

April 28, 2008
Blacksburg, Virginia

Keywords: Behavioral Programming, Action Selection, DARPA Urban Challenge, VictorTango, RoboCup, Hybrid Architecture, Unmanned Systems, Autonomous Vehicles

A Portable Approach to High-Level Behavioral Programming for Complex Autonomous Robot Applications

Jesse G. Hurdus

ABSTRACT

Research in mobile robotics, unmanned systems, and autonomous man-portable vehicles has grown rapidly over the last decade. This push has taken the problems of robot cognition and behavioral control out of the lab and into the field. Two good examples of this are the DARPA Urban Challenge autonomous vehicle race and the RoboCup robot soccer competition. In these challenges, a mobile robot must be capable of completing complex, sophisticated tasks in a dynamic, partially observable and unpredictable environment. Such conditions necessitate a behavioral programming approach capable of performing high-level action selection in the presence of multiple goals of dynamically changing importance, and noisy, incomplete perception data.

In this thesis, an approach to behavioral programming is presented that provides the designer with an intuitive method for building contextual intelligence while preserving the qualities of emergent behavior present in traditional behavior-based programming. This is done by using a modified hierarchical state machine for behavior arbitration in sequence with a command fusion mechanism for cooperative and competitive control. The presented approach is analyzed with respect to portability across platforms, missions, and functional requirements. Specifically, two landmark case-studies, the DARPA Urban Challenge and the International RoboCup Competition are examined.

Acknowledgments

There are many people without whom this thesis, as well as my graduate career, would not have been possible. My appreciation for their support, wisdom, and guidance is truly immeasurable.

My deepest thanks go to my family. My parents, Alan and Luzviminda Hurdus, have been the greatest examples of strength throughout my entire life. It is their trust and belief in my abilities as both a student and a person that have allowed me to become the individual that I am today. My older brother, Ethan, has been my secret role-model for as long as I can remember, and it is for his respect and admiration that my thirst for knowledge has always been driven.

I would also like to thank my advisers, Dr. Charlie Reinholtz, Dr. Al Wicks, and Dr. Dennis Hong, whose mentorship has proven invaluable to my growth both in and out of the classroom. I will always remember the day Dr. Reinholtz spent over an hour discussing ideas behind robotics and autonomy with me, even though I was only a sophomore in college. His ability to look past preconceptions and evaluate both people and ideas for what they truly are will always amaze me. I have never had a professor more concerned with the acquisition of actual knowledge and understanding over mere grades, then Dr. Wicks. I will always remember fondly the lessons learned in his classes and the continual appreciation he showed for his students. I have also never had a professor with a greater drive for success then Dr. Hong. I hope to always keep his talent, ambition, and love for robotics as an example before me.

Of course, I have to give enormous thanks to my graduate peers and fellow researchers. I would not trade the world for all the hours spent coding, testing, arguing, explaining, stressing, and ultimately cheering with my fellow team members. Thank you to the TORC developers, Andrew Bacha, Cheryl Bauman, Chris Terwelp, Mike Fleming and especially Ruel Faruque, my partner in Driving Behavior crime. On the university side, thank you to Patrick Currier, Jesse Farmer, Steve Cacciola, Grant Gothing, Mike Webster, Peter King, Shawn Kimmel, Dave Anderson, Tom Albieri, Dave Bass, John Weekly, Karl Muecke, Brad Pullins, and Robert Mayo, who have all been invaluable to work beside. I would also like to thank Loginn Kapitan and the SAIC team for their support of my research and our RoboCup efforts.

I would also like to thank Shawn, Myles, Tyler, and Jess, who have been there for me through thick and thin and have inspired me in so many unimaginable ways. Their compassion and love for life will always be a part of me. Finally I must thank Blacksburg and the Hokie Nation, who taught me the power of community and support, even in the face of horrible tragedy.

Contents

List of Figures

Chapter 1

Introduction

Research in mobile robotics, unmanned systems, and autonomous man-portable vehicles has grown rapidly over the last decade. This push has taken the problems of robot cognition and behavioral control out of the lab[1] and into the field. In such situations, completing complex, sophisticated tasks in a dynamic, partially observable and unpredictable environment is necessary. An approach to action selection must be used that balances appropriate elements of planning and reactivity.

Traditionally, planning and reactivity were seen to be at ends with each other, but it is now generally understood that a complimentary combination of the two is needed. Many *Hybrid Deliberative/Reactive* control architectures[24, 14, 22, 27, 33] have been developed since the mid 90's to address this problem. In the majority of these architectures we see a general trend of placing the deliberative components at a high level while the more reactive, behavior-based, components are kept at a low-level for direct actuator control. However, with advances in computing technology, a deliberative approach to low-level motion control has re-emerged [32, 29, 31]. These search based, or planning-oriented, methods [9] have proven to be very desirable from an engineering point of view for their predictability. When the scope of deliberative motion planning is kept small and is coupled with modern computing power, sufficiently fast cycle times are attainable. As a result, the scope of a behavioral control component can now be moved from low-level reflexes to higher-level decision making for solving complex, temporal

[1] Or simulated environments

problems. Chapter 2, History and Background, goes through the advancement of robotic control architectures and helps to further define a new role and scope for behavioral programming.

Within any behavior-based control system, the core problem lies in coordinating the activities of individual behaviors into rational and coherent strategies. The formulation of mechanisms to solve this problem is known as the *action selection problem* (ASP).[2] Maes defines the problem formally in [17] with the following statement.

> "How can an agent select the most *appropriate* or the most *relevant* next action to take at a particular moment, when facing a particular situation?"

Many different action selection mechanisms (ASMs) have been proposed to solve this problem and their taxonomy has been well documented by Pirjanian in [23]. There exist many relevant characteristics with which to classify ASMs, but Pirjanian finds that all ASMs can be put into either *arbitration* or *command fusion* classes. Arbitration ASMs allow "one or a set of behaviors at a time to take control for a period of time until another set of behaviors is activated." Command fusion ASMs, on the other hand, "allow multiple behaviors to contribute to the final control of the robot." Pirjanian's taxonomy classifies all ASMs as being in either one or the other of these two classes, but not both. In Chapter 3, an approach to behavioral programming is proposed that allows for the combination of both an arbitration and command fusion ASM. Such an approach preserves the benefits of both classes of ASMs.

The ultimate goal of action selection and behavior-based decision making research within mobile robotics is to build a physically embedded system that can exist autonomously in the *real world*. Action selection mechanisms that work in virtual environments are often unsatisfactory when transported to agents that must deal with real world uncertainty. It is therefore desirable to inspect the performance of any approach to behavioral programming on *real* robots performing *real* tasks. Chapter 4 and Chapter 5 present two very important case studies of behavioral programming, the DARPA Urban Challenge and the International RoboCup soccer competition. At first glance, these two real-world robotic applications are extremely different.

[2] Also known as the behavior coordination problem

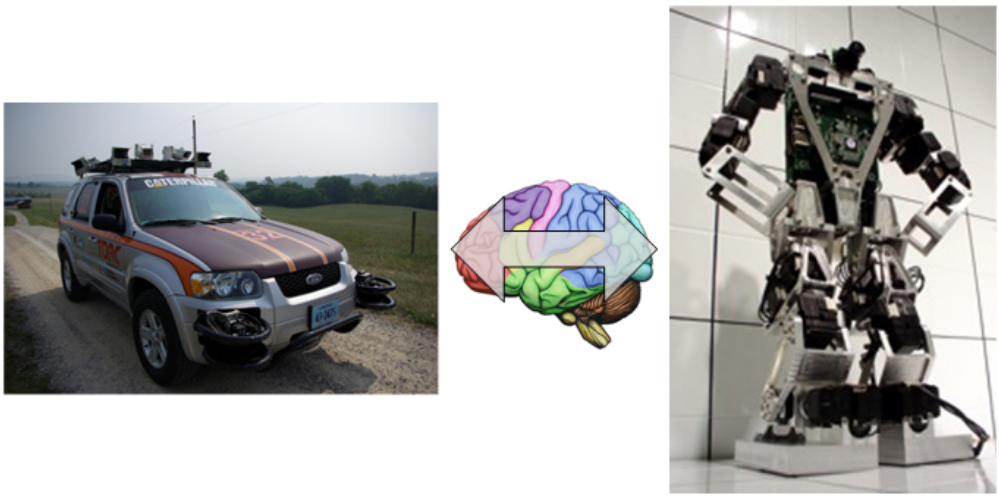


Figure 1.1: *Odin,* an unmanned ground vehicle, and *DARwIn II,* a bipedal humanoid

The DARPA Urban Challenge is concerned with building a full-sized autonomous ground vehicle capable of driving in an urban environment while negotiating traffic, intersections, and parking lots. RoboCup, on the other hand, is focused on creating a team of fully-autonomous humanoid robots capable of playing soccer. Across these two applications, the base platform is drastically different; from a 1.8 ton, 4-wheel, differentially steered vehicle to a bi-pedal, 2 foot tall humanoid robot. The goals of each robot are significantly different as well, from urban driving to goal scoring. In both of these landmark challenges, however, the core problem of a behavioral control structure is the same. Both robots must somehow balance dynamically changing desires while trying to achieve mission objectives in a real and unpredictable environment. The application and portability of the general behavior-based control approach presented in Chapter 3 is therefore analyzed with respect to these two vastly different challenges.

Finally, Chapter 6 presents the most important conclusions and observations from the work done. The most important lessons learned from the behavioral programming challenges presented by the Urban Challenge and RoboCup are summarized and any remaining questions are posed as future research topics.

Chapter 2

History and Background

In order for robots to become commonplace in the home and in industry, they must exhibit some form of *artificial intelligence*. The role of robotics can no longer be constrained to pre-programmed sequences often seen on manufacturing floors and production plants. A.I. for mobile robot control has therefore been an important area of research for over 30 years and has led to a variety of robotic control architectures. This chapter gives a brief account of the development of common robotic control architectures from the *Hierarchical* paradigm, through the *Reactive* paradigm, and into the growth of *Hybrid Deliberative/Reactive* paradigms. Looking at the emerging trends in robotic control architectures is instrumental in understanding the role and scope of present-day behavioral programming. It also serves to lay the foundation for the general approach to behavioral programming presented in Chapter 3.

2.1 What Is Mobile Robot Intelligence?

There exist many different definitions for intelligence. The appropriateness of any such definition ends up being linked strongly to the application. From a mobile robotics point of view, intelligence is primarily concerned with providing the skills and abilities for a physically embedded system to interact with and induce some change in the world. Jim Albus of the National Institute of Standards and Technology defines intelligence as

> "the ability to act appropriately in an uncertain environment, where appropriate action is that which increases the probability of success, and success is the achieve-

ment of behavioral goals." [1]

Satisfying Albus' definition of intelligence requires proficiency in several different areas including, but not limited to:

- Perception
- Knowledge Representation
- Reasoning
- Decision Making
- Planning
- Motion Control
- Manipulation

Functions such as these are considered the fundamental building blocks of mobile robot intelligence. Overall intelligence as defined by Albus is only possible when a robot is capable of performing these smaller problems well and when they are organized in certain ways. Defining and classifying all of these smaller, but necessary, building blocks is a large problem of its own. In [21], Murphy has divided the common functions of any mobile robot into three accepted primitives: *SENSE, PLAN,* and *ACT*.

2.2 The Robotic Paradigms

By dividing the common functions of robots into these categories, or primitives, it is possible to classify robotic control architectures according to various different *paradigms*. The three known robotic paradigms are structured around different methods of organizing the robotic primitives. They are known as the Hierarchical Paradigm, the Reactive Paradigm, and the Hybrid Deliberative/Reactive Paradigm. The Hierarchical Paradigm was prevalent from 1967-1990, but when the Reactive Paradigm emerged in 1988 there was a large shift away from many Hierarchical approaches. Then in 1992, the Hybrid Deliberative/Reactive Paradigm,

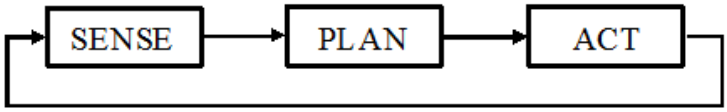


Figure 2.1: Structural organization of the Hierarchical Paradigm

which pulls many features from both of the previous paradigms, began gaining popularity and has been predominant since then. Therefore, understanding the current trends within Hybrid approaches requires knowledge of the benefits and shortcomings of the older Hierarchical and Reactive Paradigms.

2.2.1 The Hierarchical Paradigm

The origins of mobile robot control are rooted in the Hierarchical Paradigm. First devised in 1967, the Hierarchical Paradigm was born at the Stanford Research Institute (SRI) on a robot aptly named Shakey. Shakey was built on a tall, differentially steered platform, equipped with several cameras, and tasked with navigating a large room with a variety of static obstacles in it. In the end, Shakey's name was extremely appropriate as his performance was "shakey" at best. This, however, was largely due to limited sensor and computing power available at the time. Either way, Shakey pioneered the use of the Hierarchical Paradigm.

Attributes

The structure of the Hierarchical Paradigm is inherently sequential. In terms of the three robotic primitives, a Hierarchical approach will first sense the world around it, then plan a path, or set of actions, and finally act in an attempt to achieve that plan. After some time interval, the algorithm will repeat, resulting in another cycle of *sense, plan,* and *act*. A diagram illustrating the organization of a hierarchical paradigm is shown in Figure 2.1.

An important aspect of the Hierarchical Paradigm is that it is *monolithic*, in that all perception information is fused into one global data structure, or world model. During every loop iteration, the world model is updated off of what is sensed and a plan is "solved-for" using

this representation of the world. Possibly the most important attribute of any Hierarchical Paradigm lies in the planning primitive. By definition, a Hierarchical approach will take advantage of some form of global solver, or planner, to analyze all the known information about the world and deduce the best course of action for some future time period. Most commonly, this planner utilizes advanced *search* techniques to examine the set of all possible moves given a certain search space. As you can imagine, the overall performance of the robot therefore hinged largely on the performance of this planning component.

Advantages

The primary advantage of the Hierarchical Paradigm was that it pushed the development of more efficient search algorithms. In the Hierarchical approach, as the number of objects being represented in the world model increases and the number of possible actions broadens, the search space grows exponentially. Subsequently, to prevent the planning component from becoming a bottleneck, more effective search and planning algorithms were needed. The most significant breakthrough came in the development of the A* algorithm, which allows an "agent to find a sequence of actions that achieves its goals, when no single action will do." [25] The A* algorithm is widely recognized today as the most common and successful metric path planner and is used in a variety of applications, not just within mobile robotics.

Drawbacks

While improved search and planning algorithms helped to reduce the time needed to plan, it was not enough to overcome the growing complexity needed to complete even simple tasks such as navigating a room. Due to the linear nature of the paradigm, anything that changes in the environment during the planning phase is completely lost to the robot. Furthermore, the Hierarchical Paradigm is subject to a problem known as the *closed world assumption*. In order for a global planner to be successful, all relevant information about the world must be explicitly represented in the world model. If the robot ever encounters an object which it cannot classify and represent internally, it will have no way of reacting to it. This is a major shortcoming

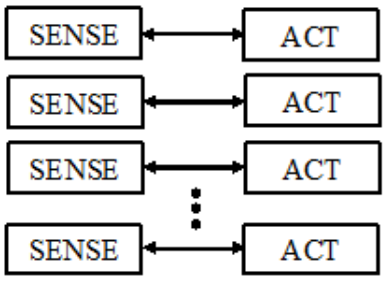


Figure 2.2: Structural organization of the Reactive Paradigm.

preventing Hierarchical Paradigms from being successful in the field.

2.2.2 The Reactive Paradigm

The Reactive, or behavior-based, Paradigm grew mainly out of dissatisfaction with the Hierarchical Paradigm. It also incorporates ideas that were emerging at the time from *ethology*, or the study of animal and insect behavior. It was clear to many researchers at the time that simple biological beings with very low computational capacity were capable of doing what Shakey could not. While the high-level planning and cognitive abilities of the Hierarchical Paradigm were neat, were they really necessary for the basic problem of robot navigation? Some researchers, Rodney Brooks of MIT in particular, didn't think so.

Attributes

The main feature of any reactive paradigm is that all actions are accomplished through the use of distinct behaviors. "Behaviors are a direct mapping of sensory inputs to a pattern of motor actions that are then used to achieve a task." [21] Mathematically, a behavior is simply just a transfer function, mapping some sensory input directly to some form of actuator output. The overall control architecture is then built up as a set of concurrently running behaviors. The result is that a robot's overall behavior emerges from the combination of behaviors operating at any given time[6]. In terms of the robotic primitives, the Reactive Paradigm leads to a much more parallel and vertical decomposition, and completely throws out any sort of planning

component, as seen in Figure 2.2.

In this type of structure, the sensing is specific to each behavior. For example, raw computer vision data may be processed in a specific way for one behavior, but differently for another. A lane following behavior might be most interested in locating the position of lane markings, whereas a vehicle-convoy behavior might be more interested in determining the location of the lead vehicle. Either behavior has no idea what the other behavior is doing or perceiving. This leads to the other major difference from the Hierarchical Paradigm: the removal of any type of global world model. In doing so, the Reactive Paradigm inherently gets rid of the closed world assumption.

Advantages

An important advantage of the Reactive Paradigm is that it allows for autonomous behavior to be built up in a similar manner to biological intelligence. In one implementation of the Reactive Paradigm known as Subsumption Architecture [6], primitive behaviors can be developed first, and then new layers of behaviors that reuse or inhibit the lower, older behaviors can be built up on top. Another very important advantage of behavior-based approaches is *graceful degradation*. If an advanced behavior fails, the lower-level behaviors will continue to operate normally. With the Hierarchical approach, on the other hand, a failure within the planning module often results in the robot's functionality grinding to a halt. One of the greatest advantages of the Reactive Paradigm is the overall speed of operation. Since each behavior consists of only some direct sense-act coupling and all behaviors run in parallel, the bottleneck of planning is removed. Rodney Brooks showed with his 6-legged robot, Khepera, that robot navigation across a room could be achieved with very low complexity algorithms when used in a behavior-based approach.

Drawbacks

While most Reactive Paradigms are computationally simple, fast, and robust when compared to Hierarchical Implementations, they are also usually much more imprecise. It is difficult to determine beforehand exactly what discrete behaviors are needed to combine and produce a rich

emergent behavior. Furthermore, reactive behaviors do not lend themselves to mathematical proofs which can definitively show that they are sufficient and correct for a task. All these factors tend to lead people into describing reactive architectures as "fast, cheap, and out-of-control." Even when the correct base behaviors are determined, exactly how they are combined has a huge effect on what emergent behavior results. In fact, the method used for combining different behaviors or selecting which behaviors are run is the distinctive feature of different behavior-based architectures and is known as the *action selection mechanism* (ASM).

As the complexity of robotic applications grew, purely reactive architectures were also found to be insufficient. Without any sort of planning component, reactive architectures eliminated any form of remembering or reasoning about the global state of the robot with respect to its environment. Not only could optimal trajectories not be calculated, but functionality such as map making and performance monitoring were lost. In total, it became very difficult to design a set of behaviors that would be wholly sufficient for completing more complex, multi-objective tasks.

2.2.3 The Hybrid Deliberative/Reactive Paradigm

With the deficiencies of the Hierarchical Paradigm in terms of low-level control addressed by the Reactive Paradigm, there emerged a need to find a way to re-incorporate some of the high-level cognitive abilities of the Hierarchical Paradigm. Techniques needed for sequencing or assembling behaviors such that a series of sub-goals can be achieved is necessary for more complex problems. The result is the Hybrid Deliberative/Reactive Paradigm which presents the idea of combining reactive behaviors with top-down, hierarchical planners used for more deliberative functions.

Attributes

An important feature of the Hybrid approach is that *deliberative* implies more cognitive abilities than just *path planning*, as it did in the Hierarchical Paradigm. Examples of these deliberative functions include performance monitoring, behavioral management, map making, and mission

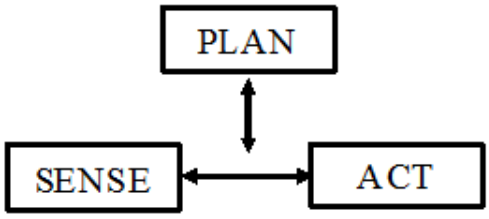


Figure 2.3: Structural organization of the Hybrid Deliberative/Reactive Paradigm

planning. In general these deliberative components are designed to run *asynchronously*, providing a set of intermediary goals for use as guidance to the lower-level reactive system. These intermediary goals should be sufficient for preventing the reactive system from making too many poor decisions as well as provide the ability to recover from being "trapped". Another important attribute of Hybrid systems is that behaviors within the reactive layer differ from traditional sense-act couplings. In the Reactive Paradigm, behaviors are purely reflexive, with no internal memory or state. Biological behaviors, however, are not just reflexive reactions, and we see in most Hybrid architectures much more complex behaviors with dynamic internal states. Returning again to the three robotic primitives, we can visualize the Hybrid Paradigm as a parallel structure where planning occurs on a slower time scale but looks ahead to a longer time horizon and helps to determine which sense-act behaviors should be running at any given time, as seen in Figure 2.3.

Advantages

The obvious advantage of the Hybrid approach is that theoretically the benefits of both the Hierarchical Paradigm and the Reactive Paradigm can be achieved. For example, sensing organization takes on a dual role. Sensor data is available to the planner components for construction of a task-oriented global world model but is also available directly to behaviors for quick reactions that do not require any type of explicit representation. The Hybrid Paradigm and the development of asynchronous processing techniques such as multi-threading have lead to much more modular designs. Individual components responsible for different functions can

run at their own frequency and then simply pass the most relevant and latest data between them. Each component is also freed to use either a hierarchical or reactive approach, allowing for a deeper abstraction of the responsibilities of individual modules within the overall control architecture. When properly implemented, a Hybrid approach can produce an appropriate balance of exploration and reactivity with an execution of planned actions, thus mimicking biological intelligence.

Drawbacks

The main challenge for any Hybrid implementation is determining where to draw the line between planning components and reactive components. Because the paradigm itself is so loosely defined, there is a wide range of differing Hybrid architectures each with their own strengths and weaknesses. It is therefore difficult to look at the drawbacks of the Hybrid approach as a whole. Of course, by incorporating both Reactive and Hierarchical elements, the Hybrid Paradigm naturally inherits the shortcomings of both other approaches. For example, the deliberative planning components within a Hybrid system are still usually subject to the closed-world assumption. However, it is usually acceptable for a robot to *think* in a closed world as long as it can still *act* in an open world.

2.3 A Closer Look at Hybrid Architectures

Both the Hierarchical Paradigm and the Reactive Paradigm were largely developed to solve one of the most well known problems for mobile robots: *navigation*. While the problem of simply getting around in the world is important, modern day robot applications require much higher levels of intelligence and complexity. Hybrid architectures present a way of addressing these problems and are therefore generally accepted as the best approach to most applications today. They allow for the layering of responsibilities from high level mission planning to low level obstacle avoidance. Individual software agents responsible for different tasks are freed to use whatever paradigm is most appropriate. According to Murphy,

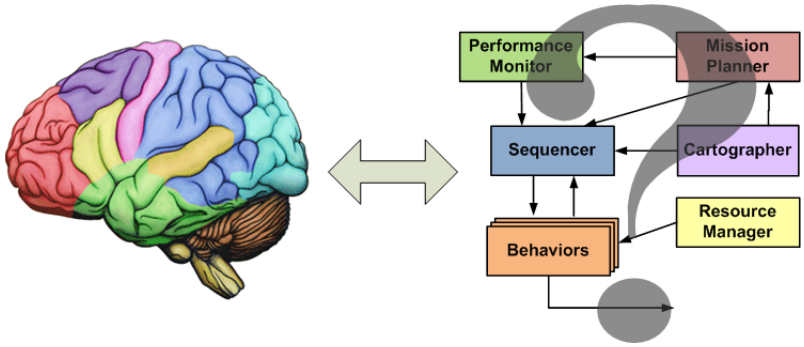


Figure 2.4: Brain-Hybrid Analogy

> "a paradigm is a philosophy or set of assumptions and/or techniques which characterize an approach to a class of problems." [21]

Individual software agents within a Hybrid architecture can therefore use the different paradigms to tackle their individual tasks. Such an approach lends itself very well to *object-oriented programming* and the portability of specialized modules to new domains. This overall structure can be loosely analogous to the way a human brain works. Large segments of the human brain can be considered to be associated with certain specific abilities, from sensor processing in the visual cortex to motor control in the brain stem. The same role-based segmentation exists in Hybrid architectures as individual software modules. The question then becomes how to organize these software modules and determine what information should be passed between them. Furthermore, how does the overall behavior emerge? Figure 2.4 illustrates this problem and analogy to the human brain.

The problem of high-level behavioral programming, the subject of this thesis, can now be seen as an *individual software agent within a larger Hybrid control architecture.* As is implied by the name, this individual module will utilize the behavior-based, or reactive paradigm to solve its specific task. To better understand the role and scope of this module however, the surrounding architecture needs to be analyzed so that the proper assumptions and requirements are clear.

2.3.1 Combining Deliberative and Reactive Control

After the success of the Reactive Paradigm in the early 90's, it was the general consensus that behavior-based methods were the "best" way of doing low-level motion control because of their speed, elegance, and simplicity. This led to most Hybrid architectures having a single division between deliberative and reactive layers. Cognitive modules responsible for mission planning, map making, performance monitoring, and behavioral sequencing were done on the higher, deliberative layer, while selected behaviors ran on the lower, reactive layer, interacting directly with the robot's actuators. Many well known hybrid control architectures were developed with this *bi-level* approach, such as the Autonomous Robot Architecture (AuRA)[5], Sensor Fusion Effects (SFX)[22], 3T[11], Task Control Architecture (TCA)[27], and Saphira[14]. In this section we will take a closer look at the Saphira architecture because it is a good example of the *bi-level* approach and also because it illustrates some important features of Hybrid control. We will then examine the newer *tri-level* approach in the context of the Hybrid architecture developed for the DARPA Urban Challenge.

2.3.2 Bi-level Approach (SAPHIRA)

The Saphira architecture was developed at SRI and is built around three important principles, *coordination, coherence,* and *communication* [14]. Coordination refers to the coordination of actuators and sensors as well as the coordination of goals and sub-goals over a period of time. Coherence implies the need for a well-maintained global perception representation, such that all planning and control components are working off the same assumptions. Finally, communication is important for building a robot that can interact with humans.

A simplified diagram of the Saphira architecture can be seen in Figure 2.5. Like most Hybrid architectures at the time, there is a very defined line between the higher-level deliberative functions and the low-level reactive system of behaviors. Within the deliberative layer, many common functions of Hybrid Paradigms are included, such as a *mission planner, cartographer, sequencer, behavior manager,* and *performance monitor.* The high-level mission planner comes

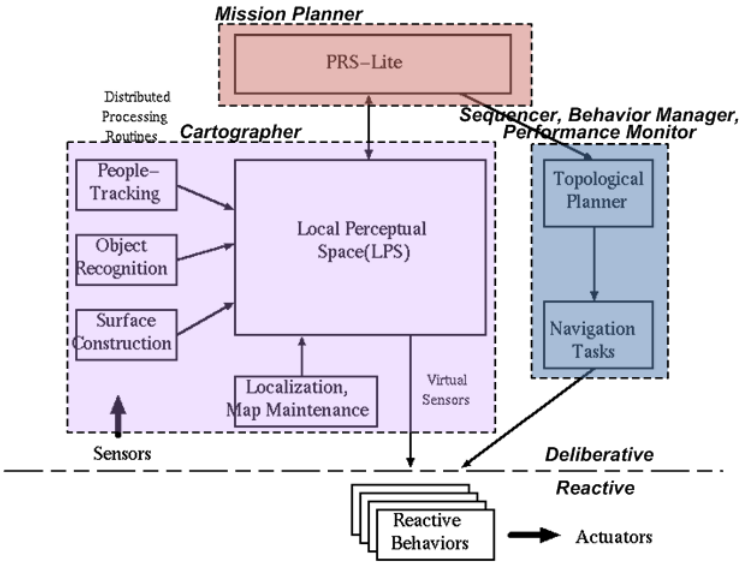


Figure 2.5: The Saphira architecture (bi-level)

in the form of the Procedural Reasoning System - Lite (PRS-Lite), which is capable of interpreting natural language voice commands into a series of manageable navigation tasks. The Topological Planner and Navigation Tasks module are then responsible for managing and sequencing behaviors as well as monitoring the performance of the robot. Within the reactive layer, the assemblage of behaviors best suited to meet the current navigation task are run and their outputs, which are different fuzzy rules, are then combined using fuzzy logic. Following convention, the *behaviors within Saphira are reflex-oriented* and largely concerned with low-level obstacle avoidance. The results of the fuzzy combination of behavior outputs are direct actuator commands. Finally, both the mission planner as well as the reactive behaviors rely on the Local Perceptual Space (LPS), which maintains an accurate world model with symbolic representations of salient features in the environment.

World Models and Virtual Sensors

A very important feature of the Saphira architecture and a growing trend in all Hybrid approaches is the re-emergence of the world model, or in this case, the LPS. Not only does the

high-level planner use this shared knowledge representation, but so do the reactive behaviors. This is very important to the idea of *coherence*. Of course, this sounds very familiar to the monolithic world models found in the Hierarchical Paradigm and should therefore come with the same problems. However, there are some very important differences which should be noted. With the advent of distributed processing, slower, more computationally intensive perception routines can run independently of the control algorithms. They then provide, via shared data structures, the latest, most relevant information. By running the perception algorithms independently, sensor errors and uncertainty can be filtered using sensor fusion over time. This approach combined with newer, more powerful processors has removed the bottleneck induced by world models in the Hierarchical Paradigm.

Another important feature is the introduction of *virtual sensors.* Reactive approaches usually rely on behavior-specific sensing, where each behavior processes raw sensor data as it sees fit. However, this is not consistent with the idea of *coherence* introduced by the Saphira architecture. In this case, the global world model provides a set of virtual sensors of which the different behaviors can "eaves-drop" on. For example, in the Saphira architecture, each behavior would have access to a people sensor, object sensor, and surface sensor. Each virtual sensor would then provide an appropriate list of people, objects, or surfaces that could be taken into account by the behavior.

2.3.3 Tri-level Approach (VictorTango)

The deliberative layer provided by Hybrid architectures such as Saphira was very important for bringing robotic applications into more cognitively challenging domains. The reactive layer was then sufficient for more time critical control problems such as low-level obstacle avoidance and could react quickly to unforeseen events and surprises. Due to the nature of the Reactive Paradigm, however, there was still no guarantee of optimality beyond the intermediary goals set by a deliberative planner. Even if the robot went in the right direction, it could easily spend extra time getting into and out of box canyons or other situations that behavior-based navigation approaches have problems with. Similar to the re-emergence of the global world

model, we see again the effect of more powerful processing technology. Path planning algorithms that once took minutes to run can now run multiple times a second. Because of this, utilizing deliberative planning techniques for low-level motion control is much more realistic.

These advancements have led to what I call the *tri-level* Hybrid approach. In a tri-level approach, there still exists a deliberative layer at the highest level, responsible for tasks such as mission planning and map making. These modules then provide intermediary goals, or waypoints, to a reactive, behavioral component that is then sandwiched by yet another low-level deliberative layer. We therefore end up with a deliberative-reactive-deliberative progression. The low-level deliberative layer is strictly responsible for motion control and sends the final output to the hardware interface. It is only concerned with events on a short time horizon and utilizes optimal planning techniques to find the best series of actuator outputs. Unlike in the Hierarchical Paradigm, where a plan is made and then the robot attempts to follow that plan until a problem occurs, a low-level path planner will continually replan, as many times a second as possible. This method of short time-horizon predictive path planning has been steadily gaining in popularity.

A good example of the tri-level approach is the control architecture developed by Team VictorTango for the DARPA Urban Challenge and is seen in Figure 2.6. In this architecture, a deliberative Route Planner is run solely on demand, when a new mission is loaded or if a road-block is encountered. This planned "route" is then provided to Driving Behaviors, a reactive, behavior-based module responsible for maintaining situational awareness while balancing the rules of the road with the goals set forth by the Route Planner. Finally, high-level motion commands are sent to the Motion Planner, which uses deliberative search methods to find the optimal path through the immediate environment. Like in Saphira, all sensor data is routed through a set of perception modules responsible for locating, identifying, and tracking specific percepts such as the road, static obstacles, and dynamic obstacles. These sensor independent perception messages are what make up the global world model and are provided to the behaviors and planning components as virtual sensors.

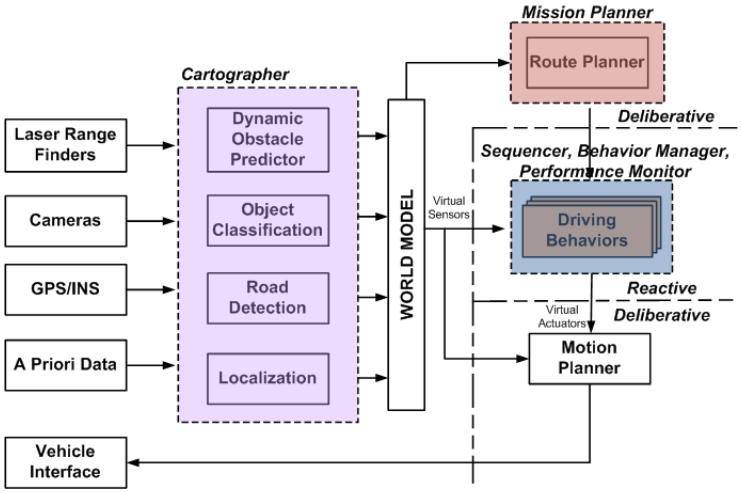


Figure 2.6: The VictorTango architecture (tri-level)

Virtual Actuators

While the Saphira architecture utilized the concept of *virtual sensors*, or high-level perception messages, the VictorTango architecture also implements the concept of *virtual actuators,* or high-level motion commands. Instead of the final output being generated by the behaviors going directly to the hardware interface, as seen in Saphira, they produce symbolic motion commands for abstract actuators. For example, in the VictorTango case, virtual actuators include a lane actuator, which specifies what lane of the road to be in, a speed limit actuator, which specifies the max speed the vehicle can drive, and a direction actuator, which specifies whether the vehicle should be traveling forward, in reverse, or if it doesn't matter. The Motion Planner component then calculates the optimal path through the environment while meeting all the requirements imposed by the virtual actuators.

The New Role of Behavioral Programming

With deliberative approaches now being used for high-level mission planning as well as low-level motion planning, the role of the reactive, behavior-based layer narrows. While this may initially seem like a bad thing for behavior-based control schemes, it is actually very useful. There still

exists a very important void in the overall intelligence of the robot that is capable of bridging the gap between high-level mission plans and low-level waypoint navigation. A software agent is needed that can provide two very important aspects of embodied A.I., *contextual intelligence* and *emergent behavior*. A reactive, behavior-based solution is well-suited to provide these important characteristics.

Contextual intelligence provides the robot with a mechanism of understanding the current situation. This situation is dependent on both the current goals of the robot, as defined by the mission planner, as well as the current environment, as defined by the relevant objects present in the world model. Such insight is important for performance monitoring and self awareness along with the ability to balance multiple goals and sub-goals. Emergent behavior is a very important trait of biological intelligence which we understand to be necessary for the success of living organisms in the real world. It allows for the emergence of complex behavior from the combination of simpler behaviors, which is important not only for individual intelligence, but cooperative intelligence within groups and multi-agent systems as well.

Deliberative approaches remain largely insufficient for providing these important traits. The reactive, behavior-based, paradigm remains the best approach for this layer of intelligence within a larger robotic control architecture. Within this context, we can now redefine the role of behavioral programming to have the following definition:

How do we determine the most appropriate, high-level actions to achieve complex mission goals and sub-goals given a dynamic, unpredictable environment?

Chapter 3

A General Approach to Behavioral Programming

As shown in Chapter 2, within the tri-level hybrid architecture, there is a distinct need for a reactive, behavioral component capable of providing *contextual intelligence* that can also be a mechanism for producing *emergent behavior.* As inputs, this software agent operates on *virtual sensors,* and as outputs, this agent controls *virtual actuators.* Virtual sensors provide a filtered view of the world and virtual actuators provide a mechanism for dictating high-level motion commands to a deliberative motion planner. It has been shown that the Reactive Paradigm is the most appropriate approach to solving this problem even though the overall goals do not coincide with traditional obstacle avoidance and robot navigation problems.

In this chapter, a general architecture and approach is presented for solving this sub-problem. A method of *action selection* is proposed that takes advantage of both *command fusion* and *arbitration* action selection mechanisms (ASMs). The ASMs discussed in this chapter are not novel themselves, but rather a novel method of combining existing ASMs is presented. The justification for using this approach stems from the specific responsibilities of a behavioral module set forth by the tri-level hybrid architecture considered in Chapter 2.

3.1 Problem Breakdown

The central problem of behavioral programming is determining at any given moment what type of *action* should be performed. Returning to Albus' definition of mobile robot A.I., a robot must "act appropriately in an uncertain environment, where appropriate action is that which increases the probability of success, and success is the achievement of *behavioral* goals." The process of deducing the most "appropriate" action is known as the *Action Selection Problem* (ASP). Unfortunately, the ability to evaluate "appropriateness" is a very complex problem and one that causes even many humans trouble. While choosing the absolutely rational, or optimal action is often impossible without seeing into the future, we can hope to select "good enough" or *satisficing* actions, as defined in [28]. According to Maes, the following requirements are needed of any ASM to produce "good enough" behavior [17].

- **Goal-orientedness** - the favoring of actions that contribute to one or several goals
- **Situatedness** - the favoring of actions that are relevant to the current situation
- **Persistence** - the favoring of actions that contribute to the ongoing goal
- **Planning** - the ability to avoid hazardous situations by looking ahead
- **Robustness** - the ability to degrade gracefully
- **Reactivity** - the ability to provide fast, timely response to surprise

In [30], the following requirements for an ASM capable of producing satisficing behavior were added.

- **Compromise** - the favoring of actions that are best for a collection of behaviors, rather than for individual behaviors
- **Opportunism** - the favoring of actions that interrupt the ongoing goal and pursue a new one

Finally, from my experiences developing ASMs for both the Urban Challenge and RoboCup, a capable ASM should also take into account

- **Temporal Sequencing** - the ability to define a necessary order for tasks and sub-tasks
- **Uncertainty Handling** - the ability to not react poorly to perception noise

It is very important to note that some of these many requirements conflict with each other. For example, persistence can be in conflict with opportunism and situatedness. Similarly, planning is in conflict with reactivity. It is therefore impossible to create an ASM which meets *all* of these requirements. Instead an ASM must attempt to *trade-off* between these requirements in a way that best fits the given application.

3.2 Taxonomy of ASMs

Being able to classify ASMs into logical groups is an important and useful exercise. Examples of such taxonomies are seen in [16], [26], and [23]. Of these taxonomies, the most complete and comprehensive is by Pirjanian in [23]. Pirjanian breaks down all ASMs as being either in the *arbitration* or *command fusion* class.

Arbitration ASMs allow "one or a set of behaviors at a time to take control for a period of time until another set of behaviors is activated." Arbitration ASMs are therefore most concerned with determining what behaviors are appropriate given the current situation. Once this has been determined it is guaranteed that there will be no conflict in outputs between the running behaviors and so no method of combination or integration is needed. ASMs within the Arbitration category are further broken down into *priority-based, state-based,* or *Winner-take-all* subclasses.

Command fusion ASMs, on the other hand, "allow multiple behaviors to contribute to the final control of the robot." Rather than being concerned with selecting appropriate behaviors, command fusion ASMs let all behaviors run concurrently, then rely on a fusion scheme to filter out insignificant behavioral outputs. Command fusion ASMs are therefore typically described of as being *flat*. Since multiple behaviors can end up desiring the same control, these ASMs present novel methods of collaboration amongst behaviors. This sort of collaboration often lends itself to multiple objective problems. For example, in the robot navigation domain, command

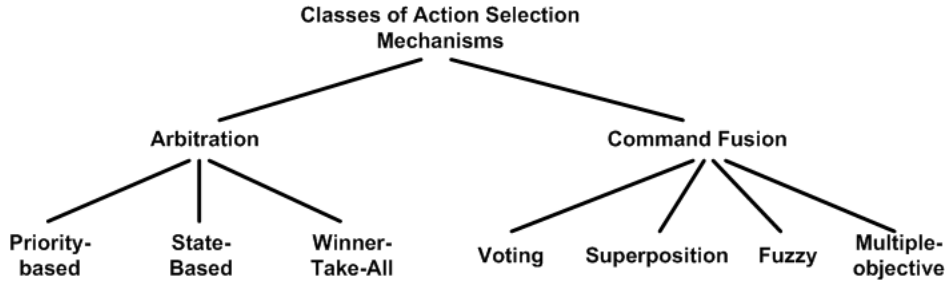


Figure 3.1: Pirjanian's taxonomy of ASMs

fusion ASMs are useful for both avoiding an obstacle and proceeding towards a goal at the *same time*. An arbitration ASM would be constrained to doing one or the other. ASMs within the Command Fusion category are further broken down into *Voting, Superposition, Fuzzy,* or *Multiple Objective* subclasses. Figure 3.1 presents the taxonomy presented by Pirjanian that has been described here.

3.3 Merging Arbitration and Command Fusion

Both Arbitration and Command Fusion ASMs have their unique strengths and weaknesses. For example, arbitration mechanisms are more efficient in their use of system resources. By selecting only one behavior from a group of competing behaviors, processing power and sensor focus can be wholly dedicated to one thing. In a flat, command fusion ASM, all behaviors must be operating at all times in order to vote for the action they prefer. As the complexity of the robot application grows, the number of behaviors needed grows, and so does the *necessary resources* in a command fusion ASM. In a hierarchical arbitration ASM, however, the library of behaviors can grow as much as it wants, but only a subset of those behaviors will ever be needed at any given moment. Of course, command fusion ASMs have their own benefits over arbitration schemes. For example, command fusion mechanisms allow multiple behaviors to simultaneously contribute to the control of the robot. This *cooperative* approach, rather than *competitive*, can be extremely useful in situations with multiple, concurrent objectives. Well known examples of arbitration ASMs include the Subsumption Architecture [6], Activation Networks [17], and

Bayesian Decision Analysis [15]. Popular examples of command fusion ASMs include Potential Fields [13], Motor Schemas [4], Distributed Architecture for Mobile Navigation (DAMN) [24], and Fuzzy DAMN [33].

In this section, a method of *merging* these two different classes of ASMs is presented. In doing so, the strengths of *both* arbitration and command fusion mechanisms hope to be preserved. **This is possible by placing an arbitration ASM in *sequence* with a command fusion ASM**. The result, in essence, is the ability to select a subset of behaviors given the current situation. Then, if multiple behaviors competing for the same output are activated, they can still be cooperatively combined using a method of command fusion. Specifically, a state-based, hierarchical, arbitration ASM is used for behavior coordination. This method utilizes a hierarchical network of Finite State Automata (FSA), which can be referred to as a Hierarchical State Machine (HSM). To integrate the outputs of the activated behaviors, almost any known method of command fusion may be used. However, the chosen method should exhibit the qualities most conducive to the specific robotic application.

3.4 Arbitration Mechanism

Using a hierarchical approach to behavior decomposition is a common practice in ethology. It allows for the differentiation of behaviors according to their level of abstraction. According to Minsky in the Society of Mind [18], intelligent beings consist of agents and agencies. All agents are organized in a hierarchy where abstract agents are built upon lower, less abstract agents. Each agent has an individual motive which it pursues by activating and deactivating lower, subordinate agents. Groups of related agents in the hierarchy are viewed as sub-systems, and the hierarchy as a whole is the overall system.

3.4.1 Hierarchical Structure

A very similar organization has been adapted here, except *agents refer to individual behaviors*. All behaviors are similarly organized in a hierarchy with more abstract behaviors higher in the

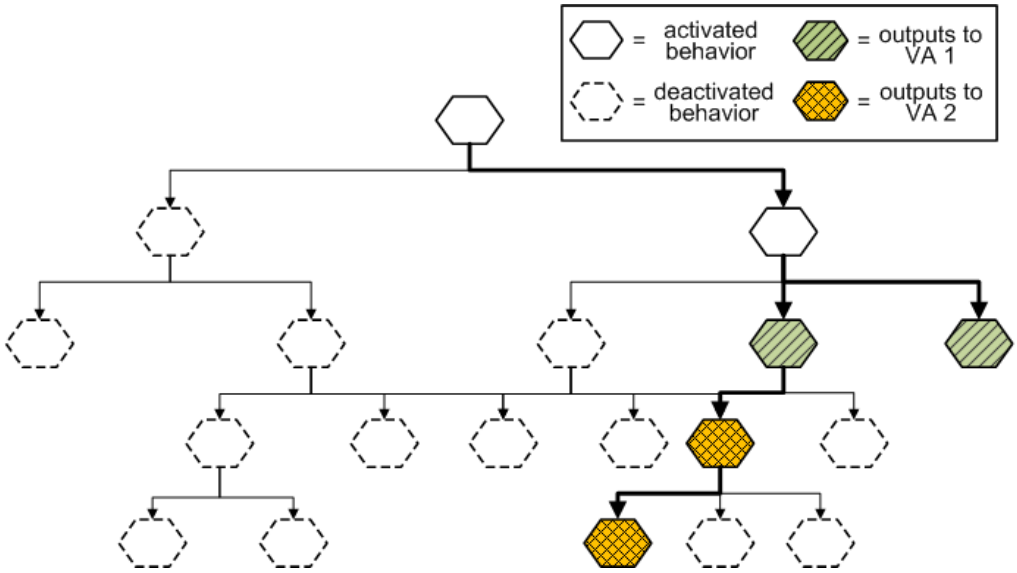


Figure 3.2: General example of a behavioral HSM

tree, and more physical behaviors lower in the tree. At any given time a subset of the total number of behaviors in the hierarchy are *activated* and the rest are *deactivated*. The activated behaviors are considered to be along the *activation path*. Each behavior, or node, in the tree is responsible for determining which of their *sub-behaviors* should be activated. This is determined by each behavior's *internal state* and is not limited to only one sub-behavior. For example, given behavior A in state X, two parallel, sub-behaviors may be activated at the same time. The result is a branch in the activation path and can be seen in Figure 3.2

We can also see from Figure 3.2 that all behaviors have implied relationships based off of their position within the hierarchy tree. Behaviors can have parent-child relationships or sibling relationships, but it is important to note that these relationships do not necessarily imply importance or priority. While some arbitration ASMs use hierarchy to determine the relevance of a behavioral output [6], this approach uses hierarchy solely as an abstraction method for task decomposition. Simply put, the primary function of the hierarchical tree is to determine what behaviors to run. Using a hierarchy allows us to logically break down a complex task into smaller, more manageable pieces.

Establishing the final output to each *virtual actuator* (VA) is therefore handled by a set of command fusion ASMs. As seen in Figure 3.2, two sibling behaviors are collaborating/competing for control of VA_1. VA_2, on the other hand, has a parent-child pair producing command messages. It is also possible for a single behavior to produce more than one VA command if it requires explicit coordination between two or more VAs. However, it is not *required* for every behavior to produce a VA command. Some behaviors, especially higher-level, more abstract behaviors may be used solely as decision nodes in the hierarchy. The internal state of these behaviors is important in determining the activation path and subsequently what lower-level behaviors will run, but do not necessarily request specific action themselves. These behaviors are seen in Figure 3.2 as activated, but not having a specific color.

Any behavior which produces 1 or more VA commands is classified as a *command behavior*. Any behavior which results in the activation of lower sub-behaviors (i.e. not a leaf node) is classified as a *decision behavior*. These classifications are *not* mutually exclusive, so it is possible for a behavior to be both a command *and* decision behavior.

3.4.2 State-Based Behavioral Structure

Every behavior is modeled as an individual state machine, or finite state automata (FSA). Individual behaviors can therefore be formally described as consisting of a set of controls states $cs_i \in CS$. Each control state encodes a control policy π_{va}, which is a function of the robot's internal state and its beliefs about the world (virtual sensor inputs). This policy, π_{va} determines what action with respect to a specific VA to take when in control state cs_i. All behaviors have available to them the same list of virtual actuators $va_i \in VA$. Furthermore, each control state has hard-coded what sub-behaviors $sb_i \in SB$ to activate when in that state.

Transitions between control states occurs as a function of the robot's perceptual beliefs, in the form of virtual sensors, or built-in events, such as an internal timer. While each behavior may have a begin and end state corresponding to the start and completion of a specific task, a single behavior, or state machine, cannot terminate itself. The higher, calling behavior always specifies what sub-behaviors should be running. Should a sub-behavior complete its state

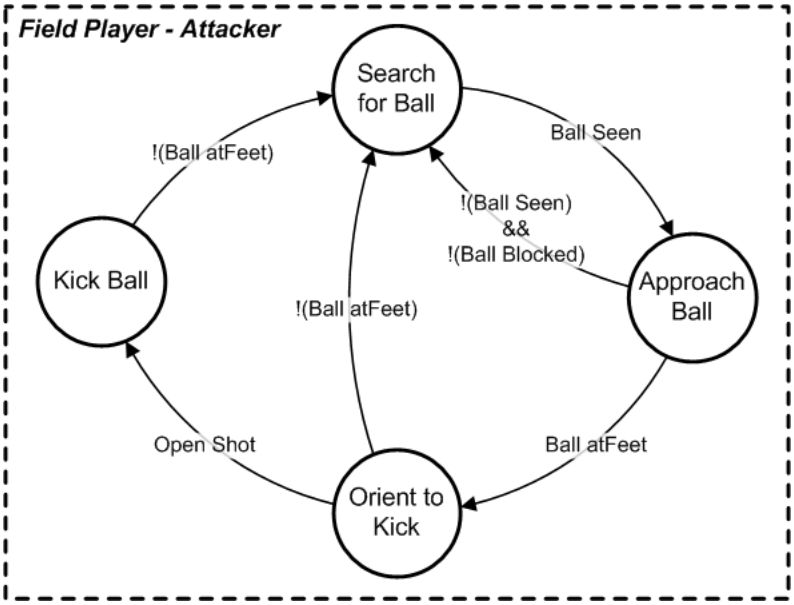


Figure 3.3: A behavioral state machine for robot soccer

sequence and have nothing to do, it will remain in an idle state and not compete for control of any VA.

A simple example of an abstract behavior used for robot soccer is shown Figure 3.3. The *Field Player- Attacker* behavior shown here is just one behavior within the overall behavior hierarchy needed for a generic soccer playing robot. It is a *decision behavior* with 4 control states and a multitude of transitions for moving between these control states. While all transitions in this example are based off of perceptual occurrences, some may require a combination of virtual sensor inputs before being evaluated to true. For example, `Open Shot` may require perceiving the goal as being in front of the robot as well as perceiving the presence of no other robots before triggering.

Of course this individual behavior is only one within a hierarchy of other more, and less, abstract behaviors. A higher-level behavior might determine the role of the robot based off of the game situation or user inputs. For example, if the team is winning significantly it might be desired to have attacking players transition to a defender role, at which point the behavior

shown in Figure 3.3 might no longer be called. On the other side, each control state shown has a selection of sub-behaviors which are activated when in that control state. Let the *Field Player - Attacker* behavior be in $cs_{ApproachBall}$, it is possible then that $sb_{BallChaser}$, $sb_{BallTracker}$, and $sb_{ObstacleAvoider}$ are activated, each with their own state machine and corresponding sub-behaviors. Since the behavior shown here is a decision behavior and not a command behavior, $cs_{ApproachBall}$ has no control policy with respect to a virtual actuator. Instead, the primary function of this behavior is to determine what sub-behaviors to run given the current situation.

From these examples we see how a HSM, and particularly the current activation path within that hierarchy, are representative of the robot's current situation. This situation is a function of the robot's *environment*, the *goals* of the robot, and the *internal states* of the robot. In total, proper construction of the HSM will result in providing *contextual intelligence* to the robot. Producing *emergent behavior*, however, is left to the Command Fusion mechanism.

3.5 Command Fusion Mechanism

As stated earlier, the hierarchical relationship between behaviors has no relevance to the likelihood of that behavior's effect on a specific VA. Once all the behaviors along the *activation path* have been defined by the arbitration mechanism described previously, their hierarchy is thrown out and they are put in a *flat* structure. Their individual outputs are then combined by a series of command fusion ASMs, with each instance corresponding to a single virtual actuator. The specific mechanism used for command fusion is not specified in this approach, and instead should be determined by the designer according to the robot application and specific virtual actuator. It is therefore possible to have one command fusion method for VA_1 of robot X, and a separate command fusion method for VA_2 and VA_3 of the same robot. This general approach to command fusion is seen in Figure 3.4.

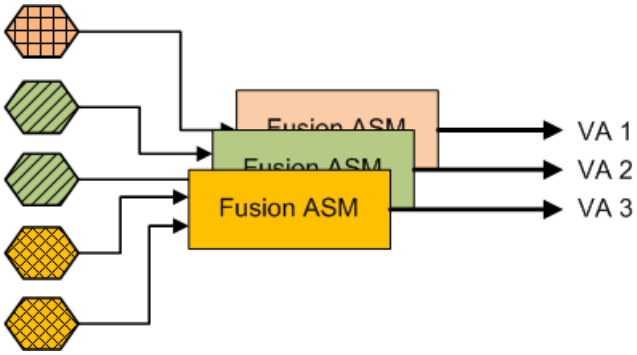


Figure 3.4: Layered Command Fusion Modules

3.5.1 Application Specific Selection

Returning to the robot soccer example presented in the previous section, let VA_1 be a vector which defines the direction and speed of a walking gait. Based on the current activation path in the HSM, the `walkToBall` behavior and the `avoidObstacle` behavior are outputting desired gait vectors. It therefore makes sense, in this behavior-based robot navigation example, to use a superposition mechanism of command fusion, such as *potential fields* or *motor schemas*. This would be the simplest way of producing the desired *emergent* behavior of approaching the ball while avoiding other robots along the way. Take now the situation where the robot is attempting to kick the ball into the opposing goal. Let VA_2 be a set of discrete kick types, `leftFoot_forward`, `leftFoot_backward`, `rightFoot_forward`, `rightFoot_backward`, etc. Just the fact that there are only a set number of discrete kick types makes a superposition-based ASM inappropriate. Instead a voting-based ASM would be much more applicable, where each behavior would vote for one type of kick, and the kick with the most votes would be selected. Taking yet another, further example, examine the behavior needed to select lanes when driving down in urban street in an autonomous vehicle. In this situation, one behavior desiring to stay in the right lane for an upcoming turn is running concurrently with a behavior desiring to pass a slow moving vehicle by moving to the left lane. Let the VA be the desired lane, and again we see that a superposition ASM is not appropriate. In this robot application, driving in between two lanes is unacceptable. Instead, a single lane should be chosen, either the left *or* the right.

We see from these examples the result of selecting different fusion ASMs. Depending on the exact mechanism chosen, completely different *emergent behavior* can result. Where the arbitration ASM was responsible for providing *contextual intelligence* to the robot, the command fusion ASM is responsible for producing *emergent behavior*.

3.6 Discussion

In this chapter, a general approach to behavioral programming that fits within the *tri-level hybrid architecture* has been presented. This approach attacks the Action Selection Problem by placing an arbitration ASM in *sequence* with a command fusion ASM. The arbitration ASM is a novel variant of existing state-based ASMs and utilizes a *hierarchical state machine* for task decomposition and behavior selection. The mechanism proposed therefore provides the robot with *contextual intelligence*. The specific command fusion ASM is not specified and should be chosen based on the robot application and corresponding *virtual actuator*. The organization of ASMs in this approach allows many typical and well known command fusion ASMs to be implemented. The selection and implementation of these command fusion mechanisms will result in the selected subset of behaviors producing *emergent behavior*. In total, the general approach presented here addresses many important problems with existing ASMs. Like with any solution, however, there are some important benefits and drawbacks.

3.6.1 Benefits

The following benefits have been identified while developing this general approach to high-level behavioral programming.

Task Decomposition The organization of behaviors in a hierarchical tree according to their level of abstraction is extremely useful for breaking down a task into manageable sub-tasks, and sub-sub-tasks that can be solved as independent solutions. Due to the fact that robotic behaviors still need to be largely hand-coded, a logical method for decomposition is very helpful in this process.

Temporal Sequencing Through the use of state machines in each behavior, the robot designer can easily imply when the order of tasks is important and when it is not. Every behavior uses a state machine to define which sub-behaviors are activated. This designer can therefore use state transitions to imply order in the completion of those lower sub-behaviors.

Behavior Reuse By taking a "divide-and-conquer" approach to behavioral problem solving, it is possible to reuse lower-level behaviors for similar problems. A sub-behavior for control state i of behavior x, can also be a sub-behavior for control state j of behavior y.

Behavior Commonalities In conventional state machines, there are many commonalities amongst different states. In the behavioral programming example, it is possible that many different behaviors would encode the same policy for a specific VA. By using a hierarchical state machine, encoding this policy in every behavior is unnecessary. Instead, a higher-level behavior allows us to define common policies only once.

Perception Requirements From a systems engineering perspective, the use of state machines is very useful because state transitions define all perception and virtual sensor requirements. By building the behavioral HSM first, a robot designer is aware of what information needs to be pulled from the environment.

Uncertainty Handling A unique property of state-based behaviors is that they can be made robust to perception noise. This is possible because state transitions are directional. The requirements for transitioning from control state A to control state B can be different then the requirements for transitioning from B to A. If there is noise in the perception data (which there usually is), defining these transitions properly can prevent flip-flopping between states.

3.6.2 Drawbacks

As expected, this approach to behavioral programming is not a ”silver bullet” solution. There are some drawbacks which should be noted.

Preprogrammed vs. Learned Individual behaviors and their relationships within the greater hierarchy must be hand-coded. As a result, determining the control policies and parameters built into each state of each behavior is a time consuming and error prone process. Testing, both in simulation and on the actual robot, is absolutely essential but not always possible. It is desirable to automatically generate or learn behaviors, or at least autonomously modify parameters and control policies based off of the robots actual experience. Such learning methods are not addressed in our approach but are being researched elsewhere [3].

Performance Measurement There exists no formal method for measuring and comparing the performance of the presented approach against other existing approaches. While ”good enough” behavior defines important functional requirements, there is no quantitative method of comparison for ”goal-orientedness”, for example. Qualitative observations are the only major source of comparison which is generally insufficient. Performance comparison of ASMs can be done in a standard simulation environment [30] or even better in real-world competitions such as the DARPA Urban Challenge. However, with non-standardized platforms, sensors, and technology, the overall performance of any team is not a good indication of the smaller behavioral programming problem. Furthermore, since the behavior hierarchy is hand-coded, different implementations of the same approach can have very different results. The overall performance, therefore, is still dependent more on the designer than the approach itself.

Chapter 4

Case Study: Autonomous Driving in Urban Environments

In November 2007, the Defense Advanced Research Projects Agency (DARPA) hosted the Urban Challenge, an autonomous ground vehicle race through an urban environment. In order to complete the course, the fully autonomous vehicle had to traverse 60 miles under 6 hours while negotiating traffic (both human and robotic), through roads, intersections, and parking lots. Out of an original field of hundreds of teams from across the globe, only 35 were invited to the National Qualifying Event (NQE) in Victorville, California. After rigorous testing, only 11 teams were selected to participate in the Urban Challenge Event (UCE). Of these 11, only 6 teams managed to finish the course, with the top three places going to Carnegie Mellon

Figure 4.1: Lineup of Finalist Vehicles in the Urban Challenge

University, Stanford University, and Team VictorTango of Virginia Tech.

In order to complete the challenge, vehicles had to contend with complex situations in crowded, unpredictable environments. A behavioral system capable of obeying California state driving laws in merging situations, stop sign intersections, multi-lane roads, and parking lots was needed. While a vehicle did not need to actively sense signs or signals such as traffic lights, right-of-way rules had to be followed as well as precedence-order at predefined intersections. This required the sensing, classification, and tracking of both static and dynamic obstacles at speeds up to 30 mph. To be successful, the vehicle had to balance goals of dynamically changing importance, traversing the course as quickly as possible while remaining a safe and "defensive" driver. The software module utilized by Team VictorTango to attack this problem employed the general approach to behavioral programming presented in Chapter 3. By examining this unique implementation and comparing it with other implementations, we can gain insight into the power and use of such a behavior-based approach.

4.1 Robotic Platform

Team VictorTango's entry, *Odin*, is a modified 2005 Hybrid Ford Escape. It is a mid-size commercial automobile that was converted completely to drive-by-wire for autonomous control. Odin's main computers, a pair of HP servers equipped with two quad-core processors each, is capable of shifting gears, applying throttle/brake, steering, activating lights/turn signals, honking the horn and even rolling down the windows. A variety of sensors provide Odin's perception, including a multitude of Laser Range Finders (LRFs), two cameras, and an Inertial Navigation System (INS)[1]. As a whole the vehicle is a very large, heavy, and powerful robotic platform capable of moving at high speeds. It is also very reliable with several safety fail-safes and the ability for a human to take control at any time. [32]

[1] An INS is composed of a differential GPS unit integrated with an Inertial Measurement Unit (IMU)

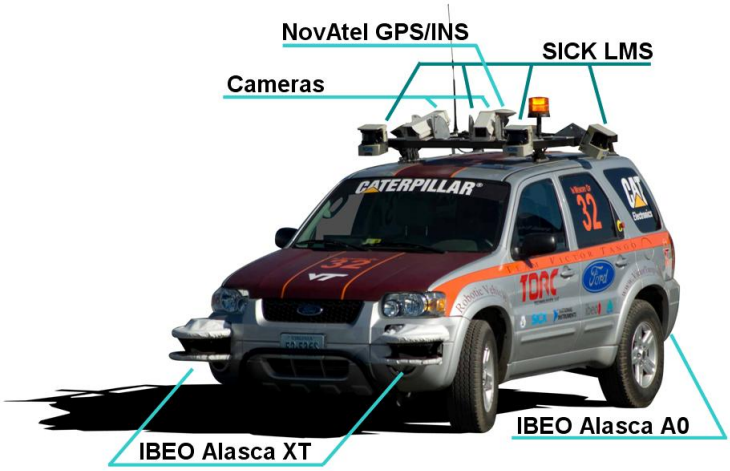


Figure 4.2: External view of Odin with sensors labeled

4.2 System Architecture

As described in Chapter 2, team VictorTango implemented a novel, tri-level, Hybrid Deliberative/Reactive control architecture as seen in Figure 2.6. This architecture follows a deliberative-reactive-deliberative progression from high-level mission planning, through behavioral control, down to low-level path planning. The architecture utilizes perception modules for producing sensor-independent perception messages in the form of virtual sensors, which are then fed to both the reactive *Driving Behaviors* module and the deliberative *Motion Planning* module. A diagram illustrating the perception structure is seen in Figure 4.3. The virtual sensors available to Driving Behaviors are described here:

Static Obstacles Objects in and around the road which have been classified as not moving and not having the potential to move. Objects such as poles, dumpsters, traffic cones, and roadblocks are listed as static obstacles.

Dynamic Obstacles Objects in and around the road which are either moving, or have the potential to move. All manned or unmanned vehicles are listed as dynamic, even if they are stopped at an intersection or on the side of the road. Dynamic obstacles also have a predicted path and lane ID associated with them. This path is a function of the road

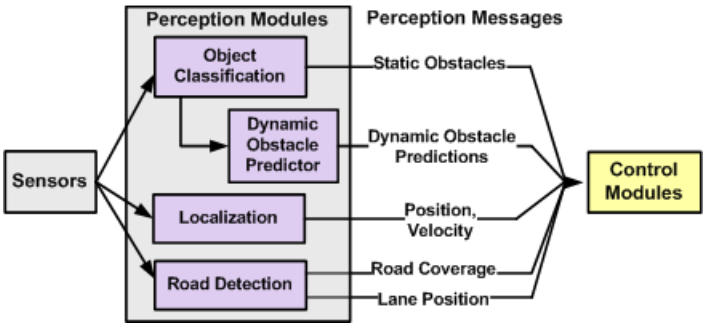


Figure 4.3: Perception and World Model Structure on Odin

coverage as well as the current and past movement of the vehicle. The lane ID is a function of the road coverage and the current position of the vehicle.

Road Coverage The drivable area around the vehicle broken down into individual lanes. Each lane consists of center points and a lane width and is organized by road and direction. Branches in the lane at intersections are also included.

Lane Position Reports the current lane position of Odin.

Local Position and Velocity State Reports the current position and velocity of Odin with respect to the "local" frame. This local frame is ground fixed, not vehicle fixed and is updated mainly from the INS system. The position of all other objects as well as the road information are given in this coordinate system.

On the control side, a deliberative Route Planner is run solely on demand, providing an optimal sequence of roads to follow to achieve the greater mission. The behavioral component, or *Driving Behaviors*, is then responsible for providing contextual intelligence, monitoring the situation at hand and taking into account the changing goals and sub-goals of the vehicle. Through a method of behavior coordination, the desired emergent behavior given the current situation must be produced through the issuing of high-level motion commands via virtual actuators. The collection of virtual actuators is known as the *Behavior Profile*. This behavior profile provides inputs to a low-level, deliberative Motion Planner that plans and re-plans

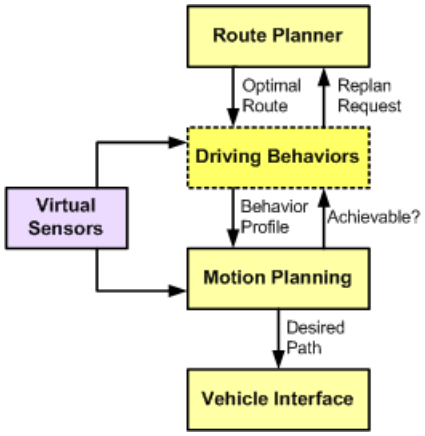


Figure 4.4: Command and Control Chain on Odin

continuously. Motion Planning maintains proper separation distances between vehicles in the same lane, and uses a trajectory search to find the optimal path to meet the behavior profile requirements while avoiding any obstacles. Finally, a desired path is sent to the Vehicle Interface which uses internal feedback loops to set proper steering, throttle, and brake percentages. A simplified diagram of this entire process is shown in Figure 4.4 with the reactive behavioral component, Driving Behaviors, highlighted. It is important to note that the command and control chain typically operates in a top-down manner, originating at the Route Planner. However, it is possible, in the event of an unforeseen roadblock, to move from the bottom-up. In this case, Motion Planning reports the behavior profile as unachievable, and then depending on the situation and the number of alternatives attempted, Driving Behaviors may request a replan from the Route Planner. This would result in an illegal, but necessary U-Turn and the continuation of the vehicle towards the ultimate mission objectives.

The virtual actuators which compose the Behavior Profile and are available for control to Driving Behaviors are described here:

Desired Velocity The desired speed of the vehicle. Motion Planning may reduce the speed of the vehicle in order to safely avoid obstacles or maintain stability, but may never increase the speed beyond the Desired Velocity.

Desired Lane This specifies the lane corridor the vehicle should be in. It is selected from the list of available lanes given by the lane position virtual sensor. Should the Desired Lane differ from the current lane position, a lane change is being commanded and Motion Planning should attempt to switch to the proper lane as soon as possible. If a Desired Lane of 0 is set, no lane maintenance is required and Motion Planning should ignore lane markings and stay only within the confines of the entire road.

Target Points A series of waypoints Motion Planning should attempt to achieve. Each target point consists of a local position and a set of behavior flags. The behavior flags are used to indicate special action that should occur at the specific target point. For example, if the *stop* behavior flag is set, the vehicle should come to a complete stop at that target point and remain stopped until a new set of target points are sent with the stop flag removed. The second behavior flag is the *heading* flag which can be used to dictate a specific heading of the vehicle when crossing over that target point. In general, lane management takes precedence over Target Points. Should they disagree, motion planning is expected to remain in the Desired Lane.

Direction Indicator This specifies whether or not the vehicle should be progressing in the forward direction or in the reverse direction. It is possible for Driving Behaviors to set this indicator to "either", in which case Motion Planning may solve to figure out which direction is optimal for meeting all the requirements being set by other virtual actuators.

Signals Specifies what turn signals to actuate along with flashing the lights and honking the horn. While this virtual actuator will not modify the path produced by Motion Planning, it is the responsibility of Driving Behaviors to control these signals in a coherent matter. For example, before passing a disabled vehicle, flashing the lights and honking the horn should be performed.

Request Replan Initiates a replan of the Route Planner. Should an unforeseen road block be encountered, Driving Behaviors should request a replan if no other method of getting around the road block exists.

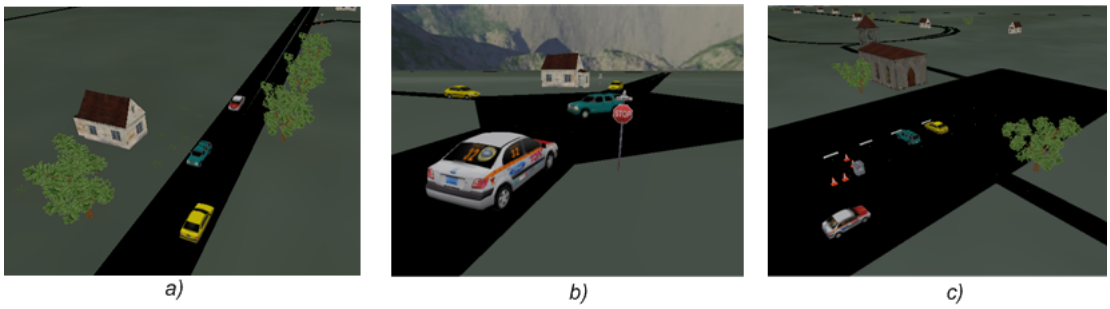


Figure 4.5: Main situational categories, the open road *(a)*, intersections *(b)*, and zones *(c)*

4.3 Task Decomposition and Hierarchy

With the available perception information, in the form of virtual sensors, and control methods, in the form of virtual actuators, defined, we can now look at the behavioral decomposition used in Driving Behaviors. Construction of the Hierarchical State Machine is dependent on the proper breakdown of required tasks and skills. Unfortunately, the entire problem of driving in an urban environment is complex with many different sets of situation dependent rules. Furthermore, it is expected that these rules be sometimes bent in order to complete more important objectives. For example, it is normally illegal to cross over a double yellow line. However, if there is a disabled vehicle on a two lane road with a double yellow line, it is within the rules to cross into the oncoming lane to pass *as long as* there is no oncoming traffic and the maneuver is properly signaled. Understanding and breaking down such ambiguities is essential.

After careful analysis of the Urban Challenge rules as well as the California state driver's handbook, the overall driving task was broken down into three main categories, the *open road*, *intersections*, and *parking lots*, or *zones*, as seen in Figure 4.5. These divisions can be defined by the structure of the road network in these areas. The open road is characterized by well defined lanes going in parallel, but opposite directions. Intersections are naturally polygons where two or more roads intersect and are well defined with entrance and exit points. Finally, parking lots have a given perimeter but no sort of lane structure. Parking spots may be defined in a parking lot, some of which the vehicle may be required to park in. Not only do these different

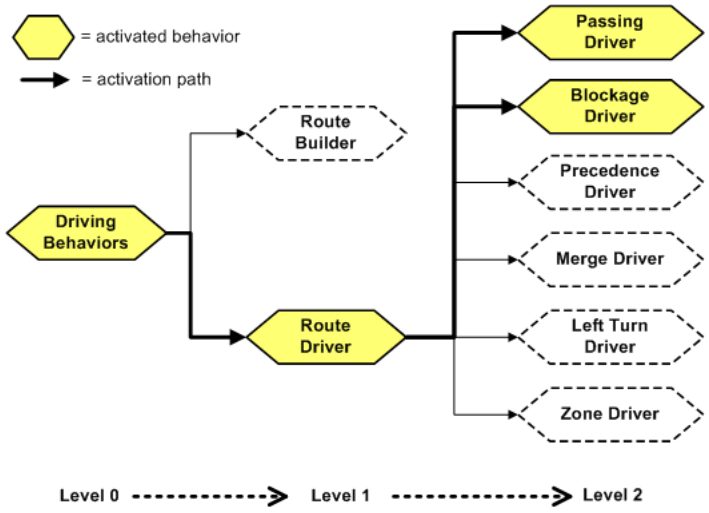


Figure 4.6: Behavior hierarchy developed for driving in an urban environment

areas have significantly different road structure, the behavior of the vehicle in these areas differs greatly. In the open road, lane changing, passing maneuvers, and U-turns are the necessary emergent behaviors. In intersections, merging, following precedence, and crossing traffic are the focus. Finally, in parking lots and zones, navigation in unstructured environments, parking, and reversing are the required skills. It is this high-level decomposition that guided the behavioral construction on Odin.

The overall HSM used for behavior arbitration on Odin is seen in Figure 4.6. The entire tree has a depth of 2, and all *command* behaviors are labeled as "drivers". Level 1 behaviors are primarily concerned with traversing the course solely on a-priori information. These behaviors utilize the given Route Network Definition File (RNDF) and other a-priori information to execute as if no surprises in the world exists. Level 2 behaviors are then responsible for handling events that cannot be planned for, such as intersection traffic, disabled vehicles, and roadblocks.

At the highest level, Driving Behaviors is always in one of only four states, *Idle*, *Build Route*, *Drive Route*, or *Shutdown*. Both *Idle* and *Shutdown* are of trivial importance, so depending on the highest level state, Driving Behaviors runs either the `Route Builder` sub-behavior or the `Route Driver` sub-behavior. The `Route Builder` behavior is activated only when a new

plan has been sent to Driving Behaviors from the Route Planner. It is neither a *command* nor *decision* behavior and therefore has no control policies and does not activate any lower sub-behaviors. Instead, it is used for pre-processing the routes to minimize calculations that need to be performed in real-time while the vehicle is driving. Most importantly, the `Route Builder` produces a complete list of Target Points that the vehicle should follow should no other traffic be encountered during the mission. This list of classified Target Points are then given to the `Route Driver` which is then activated until either the mission is completed or a dynamic replan is required.

4.3.1 Route Driver

The subsequent responsibilities of the `Route Driver` are twofold. As a *command* behavior, it has a control policy with respect to five of six total virtual actuators, π_{dir}, π_{TP}, π_{vel}, π_{lane}, and π_{sig}. These control policies are programmed such that Odin will drive the route as close as possible to the plan originally provided by the Route Planner. For example, when entering a new segment, π_{lane} will immediately move Odin to the correct lane for the upcoming exit, π_{sig} will initiate the appropriate turn signal once within a certain distance, and π_{TP} will set a stop flag at the exiting Target Point if the road network dictates that the vehicle must stop (i.e. a stop sign is present). If the vehicle is not *required* to stop (i.e. a yield situation), then the Route Driver will not set a stop flag, and it is the responsibility of a more specific intersection behavior to stop the vehicle due to traffic.

As a *decision* behavior, the `Route Driver` must determine what sub-behaviors to activate. The available sub-behaviors are the `Passing Driver`, `Blockage Driver`, `Precedence Driver`, `Merge Driver`, `Left Turn Driver`, and `Zone Driver`. These drivers are chosen based on the internal state of the `Route Driver`. All of the `Route Driver`'s states are classified as either *open road, intersection,* or *zone*. In *open road* states, the `Passing Driver` and `Blockage Driver` behaviors are activated. In *intersection* states, depending on the type of intersection, some combination of the `Precedence Driver`, `Merge Driver` and `Left Turn Driver` is activated. Finally, in *zone* states, the `Zone Driver` is activated. Figure 4.7 shows the state diagram of

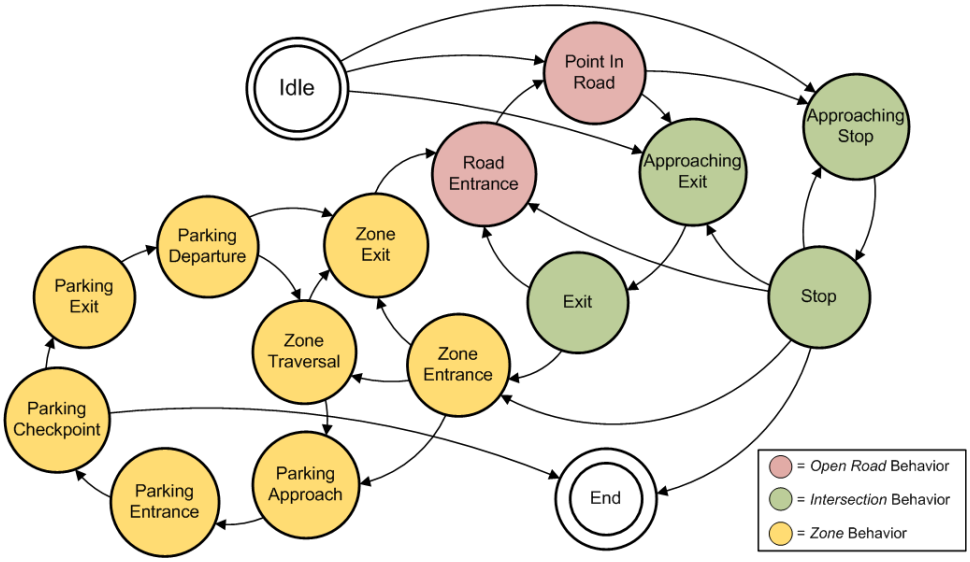


Figure 4.7: State Diagram of the `Route Driver`

the `Route Driver` and the classification of states as either open road, intersection, or zone.

From this state diagram, we can see some important behavioral traits. For example, before every intersection, which are defined as being either *stop* or *exit* intersections, there is a corresponding *approaching* state. The transitions of the state diagram show that it is impossible to enter a stop or exit state without first moving through the approaching state which is entered once the vehicle is within a certain distance threshold to the upcoming intersection. As you can see in the Figure 4.7, these approaching states are defined as *intersection* states and not *open road* states, meaning the `Passing Driver` and the `Blockage Driver` are no longer activated. This prevents the vehicle from trying to pass a traffic queue for being too slow, or mistaking the traffic queue as a roadblock and commanding a dynamic replan.

4.3.2 Passing Driver and Blockage Driver

Unlike the `Route Driver`, both the `Passing Driver` and `Blockage driver` are solely *command* behaviors and not *decision* behaviors. This does not imply that the these behaviors are void of decisions, but simply that they are leaf nodes and do not activate any lower sub-behaviors.

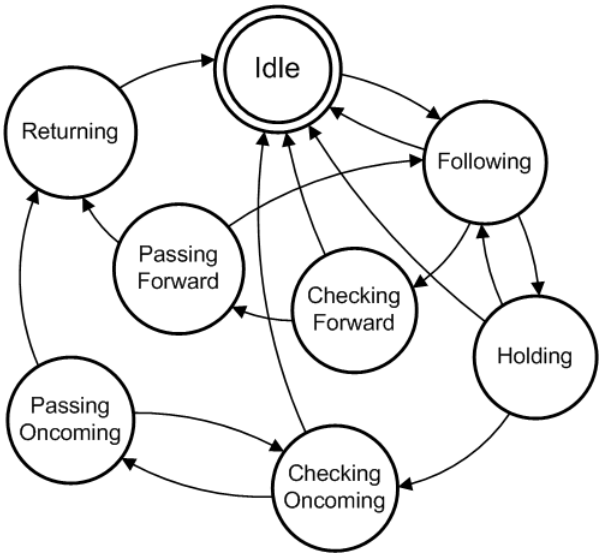


Figure 4.8: State Diagram of the `Passing Driver`

Therefore the internal states of these behaviors are used primarily to modify control policies. The available control policies to the `Passing Driver` are π_{vel}, π_{lane}, and π_{sig}. The available control policies to the `Blockage Driver` are the same as the `Passing Driver` with the addition of π_{replan}, which gives this behavior the ability to request a replan from the Route Planner module.

The `Passing Driver` is concerned with getting around slow moving or disabled vehicles. It is therefore responsible for monitoring other vehicles in the near vicinity, deciding if a pass is necessary, and executing this pass in a safe and legal manner. Awareness of the roads is necessary as the `Passing Driver` must distinguish between passing in an oncoming lane and passing in a forward lane, and subsequently check the appropriate areas for traffic. The `Passing Driver` does not maintain knowledge of the overall route however, so it is the responsibility of the `Route Driver` and the command fusion ASM to overrule or deactivate the `Passing Driver` if a pass is initiated too close to an exit or intersection. The state diagram of the `Passing Driver` is shown in Figure 4.8.

Important behavioral traits can again be pulled from the breakdown of states seen here.

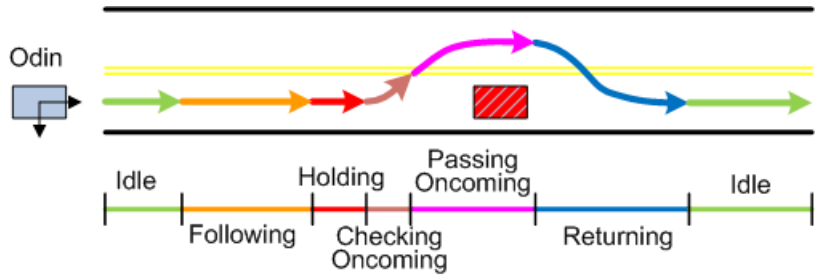


Figure 4.9: State sequence for passing a disabled vehicle in an oncoming lane

Like the pre-emptive *approaching* states in the `Route Driver`, we have pre-emptive *checking* states that must be entered before commanding an actual pass. These checking states are differentiated by whether or not a forward lane of travel exists, which in turn defines what the behavior expects other vehicles to be doing in that lane. We also see the addition of an extra *holding* state when the only available passing lane is an oncoming lane. This is a good example of the power of state machines for *temporal sequencing*. Before Odin can cross a double yellow-line, a certain amount of time must pass before the blocking vehicle is classified as disabled. During this time, π_{sig} flashes the headlights and honks the horn. Once the vehicle is determined to be disabled, the `Passing Driver` transitions to the *checking oncoming* state where π_{sig} is now used to activate the turn signal. π_{vel} and π_{lane} are used to begin a "creeping" maneuver to give the sensors ample view down the oncoming lane and ascertain whether or not a passing maneuver is possible. If there is a large enough gap in traffic, then the `Passing Driver` moves to the *passing oncoming* state and switches to the oncoming lane. Finally, after moving past the disabled vehicle, the *returning* state moves the vehicle safely back to the travel lane. The total state sequence followed to pass a disabled vehicle in an oncoming lane is shown in Figure 4.9.

Unlike the `Passing Driver`, which is primarily concerned with *dynamic obstacles*, the `Blockage Driver` is concerned with *static obstacles*. Static obstacles such as cones and barrels may be used to block off individual lanes as well as the entire road. Should only a single lane be blocked, the overall behavior should be similar to passing a vehicle. However, a method is needed to evaluate when all lanes have been blocked and to subsequently request a replan and

3-pt U-turn maneuver. The `Blockage Driver` therefore has control policies with respect to the same virtual actuators as the `Passing Driver` but with the additional control policy, π_{replan}, for requesting a replan.

The `Blockage Driver` maintains an internal list of available lanes in the present segment. These lanes are ordered such that the immediate lanes to the vehicle's left or right are chosen first. All the forward lanes are also given a priority over any oncoming lanes. Each time Motion Planning reports a lane corridor as unachievable (due to static obstacles), this list is updated by removing the specified lane and thereby bumping up the priority of all other lanes. All lanes are tried in turn, and when no more available lanes exist, either in the forward or oncoming direction, the `Blockage Driver` enters the *replan* state. This updates the Route Planner with the appropriate blockage information and results in all behaviors being reset while a new route is generated. The state diagram of the Blockage Driver can be seen in Appendix A and is provided as reference.

4.3.3 Precedence Driver, Merge Driver, and Left Turn Driver

To handle intersections, three different *command* behaviors (Precedence, Merge and Left Turn) are activated in the *approaching stop, stop, approaching exit,* and *exit* states of the `Route Driver`. As *command* behaviors, their internal states are used to modify their control policies and do not activate any lower sub-behaviors.

The `Precedence Driver` is responsible for maintaining precedence order at intersections with more than one stop sign. Upon arrival at the stop sign, this behavior examines all other stop points that are a part of the same intersection. If another vehicle is currently occupying one of these spaces, it is placed in a queue in front of Odin. The `Precedence Driver` then enters the *waiting for turn* state where it monitors the occupied positions until they have all been vacated. Once it is Odin's turn, the `Precedence Driver` continues to hold the vehicle back until the intersection itself is clear of any traffic. Once satisfied, control is relinquished back to the `Route Driver` or a separate intersection behavior. The Command Fusion mechanism used to handle this passing of control is discussed further in section 4.4. The only control policy

encoded into the `Precedence Driver` is π_{TP}. By producing a set of target points with a high urgency and the stop flag set, the `Precedence Driver` is able to overrule the `Route Driver` and hold Odin at a stop point. Since the `Route Driver` assumes no other traffic, it would want to proceed immediately after coming to a complete stop. It is therefore the responsibility of the `Precedence Driver` to hold the vehicle in place until it is also Odin's turn to proceed. The state diagram for the `Precedence Driver` as well as all other behaviors used on Odin can be found in Appendix A.

The `Merge Driver` is responsible for determining when there is space to enter, or cross lanes of moving traffic. This driver uses only one control policy, π_{vel}, to either hold Odin back when there is not enough space to merge, or possibly increase the merging speed when a smaller gap is detected. Another important trait of both this behavior and the `Left Turn Driver` is the ability to commit to merging in the face of noisy perception data. It is quite common in busy intersections to determine that there is space for a merging behavior, but due to a tight turn, static obstacle, obstructing hill, or some other condition unknown to Driving Behaviors, have the gap shrink or disappear before the merge has been completed. While it is desirable to cancel the merge in this situation, it is not acceptable to come to a complete stop in the middle of an intersection. In many cases it is in fact safer to continue with the maneuver and complete the merge then to freeze in the line of moving traffic. Therefore a "point-of-no-return" is continually calculated from the stopping distance needed to come to a complete stop without protruding into traffic lanes. Once this "point-of-no-return" has been crossed, the behavior is committed to the maneuver and will not stop itself.

Finally, the `Left Turn Driver` is very similar in responsibility to the `Merge Driver`. The `Left Turn Driver` maintains two control policies, π_{TP} and π_{vel}, and is used when making a left off of a main road across traffic. Functionally, it interacts with the `Route Driver` the same way that both the `Precedence Driver` and `Merge Driver` do, holding Odin back until a sufficient gap has been detected in traffic. Furthermore, another "point-of-no-return" is used to prevent the vehicle from stopping mid-maneuver due to perception noise.

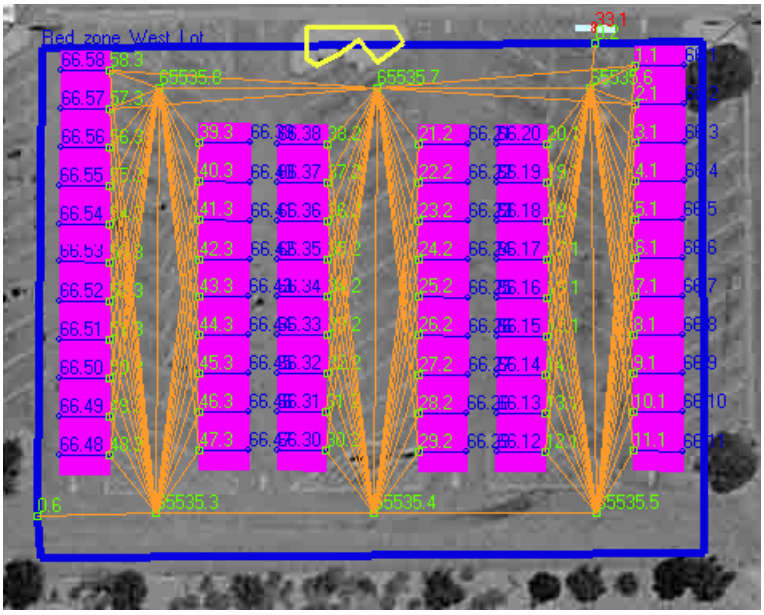


Figure 4.10: Network of control points used for navigating zones

4.3.4 Zone Driver

Keeping consistent with the fact that all level 2 behaviors in the hierarchical tree are responsible for dynamic, unforeseen events, the majority of zone behavior is actually performed by the `Route Driver`. It is for this reason there are so many zone states within the `Route Driver` as seen in Figure 4.7. If no traffic is encountered in a zone, it is possible for the `Route Driver` to perform all necessary parking behavior. All necessary Target Points for traversing a parking zone, parking in a defined spot, and exiting the zone are determined from a-priori information. Based on the organization of parking rows, and the placement of exits and entrances in the zone, a network of *parking control points* are defined as seen in Figure 4.10. A Dijkstra's search algorithm is then used during the route building phase to create a sequence of Target Points needed to safely enter the zone, park in all required spots, and exit the zone. These Target Points are then built into the route passed to the `Route Driver` in level 1 of the hierarchy.

Should, however, obstacles within the unstructured zone environment make it impossible to hit all the predefined Target Points, it is the responsibility of the Zone Driver to find a way

around. This is done by disconnecting a segment of the control point graph and re-searching for a new sequence of control points. To enable this capability the Zone Driver maintains the following control policies: π_{dir}, π_{TP}, and π_{vel}. It is important to note that Driving Behaviors is not responsible for any obstacle avoidance or traffic behavior in zones. Instead, Motion Planning controls the path of Odin to stay to the right of oncoming traffic, navigate into a parking spot, and reverse out safely. This is largely due to the unstructured environment of a zone. Without well defined lanes, it is impractical for Driving Behaviors, without any knowledge of the size and mobility constraints of the vehicle, to estimate what is possible to traverse. Rather, Motion Planning can report a behavior profile as unachievable in a zone, causing the Zone Driver to kick in and initiate a new control point search.

4.4 Command Fusion Mechanisms

With all the *command* behaviors defined as individuals, it is necessary to look at how these behaviors interact as a group. Since control policies are classified by *virtual actuator,* the main interaction between behaviors occurs when two or more control policies are outputting to the same virtual actuator. For example, when π_{lane} of the Passing Driver disagrees with π_{lane} of the Route Driver, some form of resolution is needed. When this situations occurs, it is the responsibility of the command fusion mechanism to produce the final desired lane that will be bundled in the *behavior profile*.

As described in section 3.5, the HSM is responsible only for task decomposition and determining what behaviors should be running at any given time. Once these behaviors have been selected, they are placed in a *flat* organization. A command fusion mechanism is then used for *each* virtual actuator to determine the final output. This command fusion mechanism can utilize many existing techniques [4, 6, 13] or implement a novel method. What dictates the choice should be the robot application and the specific virtual actuator. For example, virtual actuators that have only a set number of possible commands do not lend themselves well to superposition based ASMs.

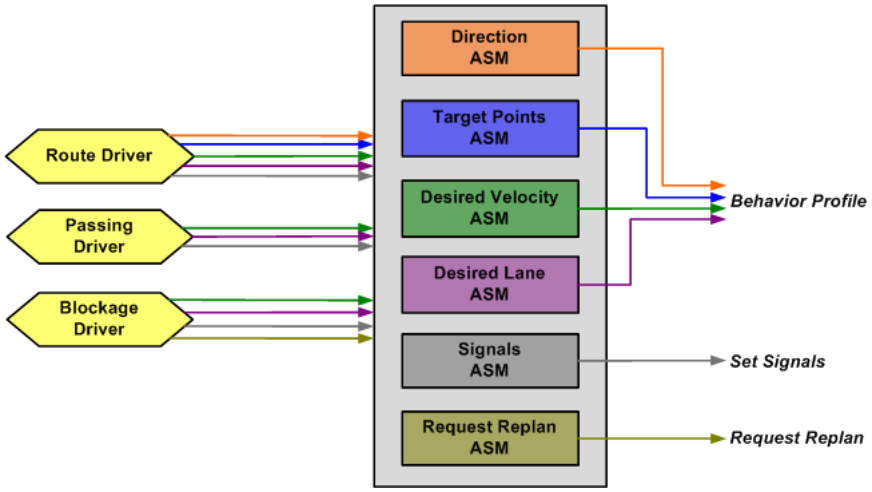


Figure 4.11: Command Fusion ASMs used on Odin

Figure 4.11 illustrates the command fusion structure used in the DARPA Urban Challenge. For simplicity, the activation path depicted in Figure 4.7 is reflected here. In this situation, only the Route Driver, Passing Driver, and Blockage Driver are activated. All of these behaviors have their own set of control policies, some of which overlap with each other. When this occurs the respective command fusion ASM resolves the conflict. In this figure, the *Set Signals* and *Request Replan* outputs are separated because they are not actually bundled into the *Behavior Profile* and sent to Motion Planning. *Set Signals* is sent directly to the Vehicle Interface and *Request Replan* is actually sent upwards to the Route Planner.

For the Urban Challenge, it was determined that the same fusion mechanism was appropriate for *all* virtual actuators. The chosen mechanism uses a modified *voting-based* policy. Due to the rule heavy nature of urban driving, and the presence of well defined individual maneuvers like passing, a non-superposition based ASM was needed. Continuing the example from above, should the Route Driver and the Passing Driver desire to be in different lanes, it is not acceptable to drive in between two lanes. Instead, one behavior should take priority and take complete control of that virtual actuator. To make this possible, each behavior has encoded in all of its control policies a variable *urgency value*. This value indicates how badly that behavior feels its control policy should be adhered to. The command fusion mechanism then has the simple

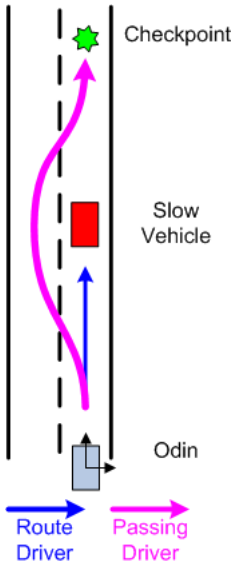


Figure 4.12: Interaction between the `Route Driver` and `Passing Driver`

responsibility of selecting the control policy with the greatest urgency. When only one behavior has an active control policy with respect to a virtual actuator, this decision is even simpler.

An example of this type of selection mechanism can be seen in Figure 4.12. In this example, the Route Driver is commanding to stay in the right lane for the upcoming checkpoint. The Passing Driver, however, is only concerned with proceeding as quickly as possible down the road. With the presence of the slow moving vehicle in the right lane, the Passing Driver would like to perform a passing maneuver. Since the checkpoint is still far enough down the road, we see in this case that the Passing Driver is able to take control and maneuver around the vehicle. Should the checkpoint be closer, however, it is the responsibility of the Route Driver to overpower the Passing Driver and prevent the pass. Another example of this behavioral interplay is seen at stop signs. In this situation, the Route Driver is concerned only with coming to a complete stop, as defined by the rules-of-the-road, and then continuing the mission. It is not concerned with any other traffic. The responsibility therefore falls to the Precedence Driver and Merge Driver to output Target Points with the stop flag set at a higher urgency. Once it is Odin's turn to proceed and there is enough room to merge, these behaviors simply lowers their

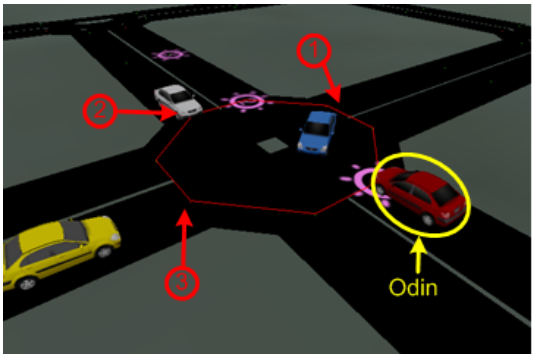
urgency's and the Route Driver resumes control, moving the vehicle into the intersection.

4.5 Performance Results

As mentioned in Chapter 3, the evaluation of a behavioral structure such as this can be quite difficult. With the general complexity of a fully autonomous mobile robot system, there are a large number of factors that might effect the overall performance. With the help of simulation, however, it is possible to try and isolate the action selection process encapsulated in Driving Behaviors with repeated tests. This process is not only important for evaluation but also for tuning and debugging the control policies and state transitions within each behavior.

In this respect, it is possible to gain some insight into the performance of this *specific implementation* of the behavioral programming approach presented in Chapter 3. From repeated testing in simulation, and the performance of Odin at the actual competition, it is acceptable to say that Driving Behaviors is capable of handling all requirements defined by the DARPA Technical Evaluation Criteria[8]. These criteria are broken down into four major categories, Basic Navigation, Basic Traffic, Advanced Navigation, and Advanced Traffic. Basic Navigation and Traffic skills include passing over all mission-defined checkpoints, the appropriate use of lanes on multi-lane roads, obedience to speed limits, 4-way stop sign behavior, queuing behavior, passing of disabled vehicles in an oncoming lane, and 3-pt U-turns. Advanced Navigation and Traffic skills include parking zone behavior, dynamic re-planning in the case of unforeseen roadblocks, merging into moving traffic, performing left turns across traffic, and handling traffic jams within an intersection.

An example of the technique used for evaluating 4-way stop behavior is shown in Figure 4.13. In this example, Odin is given a mission proceeding through a 4-way stop intersection. The arrival time of three other vehicles at the same intersection is then strictly controlled to require different behavior of Odin. For example, in one test as Odin comes to a stop, there is another vehicle just arriving at position 1, and two other vehicles already stopped at positions 2 and 3. In this situation it is expected that Odin should traverse the intersection 3rd. These results



Intersection Position			Expected Result
1	2	3	
A	*A*	*A*	1st
A	*A*	*S*	2nd
A	*A*	*E*	2nd
A	*S*	*S*	3rd
A	*S*	*E*	3rd
S	*S*	*E*	4th
S	*S*	*S*	4th

A = Vehicle Arriving, S = Vehicle Stopped, E = Vehicle Entering Intersection

Figure 4.13: Validation of 4-way stop behavior in simulation

can be easily verified and behaviors subsequently tweaked until the expected result is always achieved. It is then possible to add in unexpected results such as another vehicle proceeding out of turn or stopping in the middle of the intersection. The expected action of Odin is still easily verifiable as being either correct or incorrect.

Another example of behavior evaluation in a much more complex situation is seen in Figure 4.14. This difficult situation was actually encountered during a live test at the National Qualifying Event (NQE). This test involved handling a series of cars parked on the side of a small two lane road in a residential area. The road was then further cluttered with cones and traffic hazards along the middle of the road. The cars parked along the sides of the road were also well within the defined lanes, so differentiating these vehicles from normal traffic is extremely difficult. Several problems were found with the initial interplay of behaviors in this situation as well as the interplay between Driving Behaviors and Motion Planning that caused Odin to get stuck. However, with repeated simulation testing, these issues were resolved and the vehicle was subsequently able to pass through the gauntlet without problem.

It is important to note, however, that simulation testing is not sufficient for the full evaluation of behaviors. The simulation environment used by team VictorTango was not capable of simulating noisy perception data. Therefore, testing of intersection behavior, for example, in the face of disappearing and reappearing vehicles due to occlusions or sensor noise is not possible. This must be done via live testing on the actual vehicle which is obviously more difficult

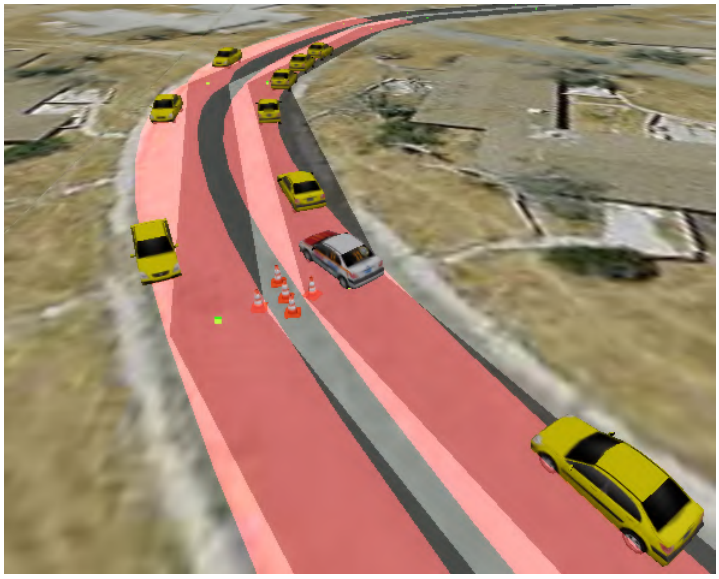


Figure 4.14: Validation of the "gauntlet" behavior in simulation

to coordinate and less frequent. For this purpose, detailed logging of the internal processes of all behaviors is required along with a useful method of playback for post-processing. Tools for this purpose were developed by team VictorTango and used extensively after live tests. These tools allowed the exact virtual sensor messages to be re-produced and fed back into Driving Behaviors for very controlled evaluation and debugging.

In total, the behavioral Hierarchical State Machine and Command Fusion mechanisms presented in this chapter were sufficient for producing an excellent performance at the Urban Challenge Final Event. Odin placed third overall, completing the course and all of its rigorous tests well within the six hour limit and only minutes behind the leaders. After post-processing all the recorded data from the final race and examining hours of video, it was determined that Driving Behaviors made *no incorrect decisions* throughout the *entire* course of the race. An image of Odin at the finish line podium is seen in Figure 4.14.

Figure 4.15: Odin at the finish line next to Stanley of Stanford and Boss of Carnegie Mellon

4.6 Lessons Learned

While Odin performed very well at the Urban Challenge, there are still some important lessons to be learned and areas for improvement in the behavioral implementation. The most important lesson has to do with the reliance on the behavioral designer and the importance of exhaustive testing. The greatest strength of the general approach is that it provides an intuitive method for task decomposition while maintaining the qualities of command fusion for emergent behavior. These benefits, however, are still dependent on the ability of the designer to predict almost all complex situations that the robot might encounter, and then to test them repeatedly both in simulation and in live tests. Without this, construction of an effective behavior-based software module responsible for handling complex problems is nearly impossible.

For example, all individual behaviors are designed to be independent, with no direct knowledge of the activity in other concurrent behaviors. However, in order to produce the right emergent behavior from the interplay of control policies, higher knowledge is needed of the potential urgency values for each behavior. In the Urban Challenge implementation, the designer must be aware that the baseline urgency for the Route Driver is x so that they can program

the potential urgency of the Passing Driver to be $y > x$ when needed. Otherwise, the Passing Driver would never take control. Modification of the control policy in one behavior with respect to urgency can therefore have large effects on the resulting action produced by another behavior. As a result, the designer must be cognizant of what behaviors will be activated at the same time, and design those behaviors accordingly. In total, while all behaviors are modular software agents, there immediate re-usability across different platforms and applications is limited. Control policies and transition requirements must inevitably be tweaked to produce the right emergent behavior. While machine learning and optimization techniques could potentially be applied here, it is difficult to create a standard formulation of parameters and performance criteria for widespread use of such methods across all behaviors.

These drawbacks are largely due to the selected command fusion ASM. The *urgency* based voting policy used for all virtual actuators in the Urban Challenge is very simple in nature and subsequently inflexible. A different approach might result in less hard-coding for the designer. Examples such as fuzzy logic and multi-valued behavior outputs would allow for better *cooperation* amongst behaviors compared to the very *competitive* method in place. While using such a methods might alleviate some of the intricacies of setting urgency's, they could potentially reduce the overall robustness and predictability of the behavioral system. When driving a full-sized commercial vehicle through crowded streets, however, such a trade-off is not acceptable.

Chapter 5

Case Study: Humanoid Robot Soccer

The very first RoboCup conference was held in July of 1997 in Nagoya, Japan. Since then, it has grown into a multi-national research and education initiative centered on an annual conference and robotic soccer competition. The goal is to use the standard problem of robot soccer to foster the growth of artificial intelligence and mobile robotics research. The landmark challenge presented by RoboCup is to develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team by the year 2050. This requires significant advances in the areas of multi-agent collaboration, strategy acquisition, real-time reasoning, behavioral programming, bi-pedal locomotion, machine vision, and sensor fusion, to name a few. While it is unlikely that this landmark project will be accomplished in any near term, the idea of soccer as a standard arena for mobile robots has been widely accepted. It is estimated that more than 500 teams consisting of 3,000 scientists from 40 countries will participate in RoboCup 2008 in Suzhou, China, making it the largest competition in the project's history.

The Robotics and Mechanisms Laboratory (RoMeLa) of Virginia Tech has developed a team of fully autonomous humanoid robots for entry in the kid-size humanoid division. In this division a team of 3 fully autonomous humanoid robots must play the game of soccer against another team of robots. All sensing and processing must be performed on-board, and wireless transmission may be used only for communication amongst individual players. All sensing must

Figure 5.1: DARwIn IIa and IIb competing at RoboCup 2007

be roughly equivalent to the capabilities of a human, prohibiting the use of active sensors that emit light, sound, or electromagnetic waves. Furthermore, external sensors such as cameras and microphones may not be placed in the legs, arms, or torso and must have a limited field of view similar to a human. In order to qualify for competition, robots must be able to localize an unknown ball position, walk to the ball while maintaining stability, localize a goal and position around the ball for kicking, kick the ball towards the goal, and autonomously detect and recover from a fall. To perform well in competition, robots must also be able to defend against other teams attacks, dive to block kicks if designated as a goalie, avoid contact with other robots, and work strategically as a team. An example of a typical game-time situation can be seen in Figure 5.1.

Like the Urban Challenge, each individual robot must be able to handle complex situations in an unpredictable and noisy environment. A behavioral system is needed that can balance multiple goals of dynamically changing importance such as scoring, defending, and maneuvering. Therefore, a method for providing *contextual intelligence* and the ability to produce *emergent*

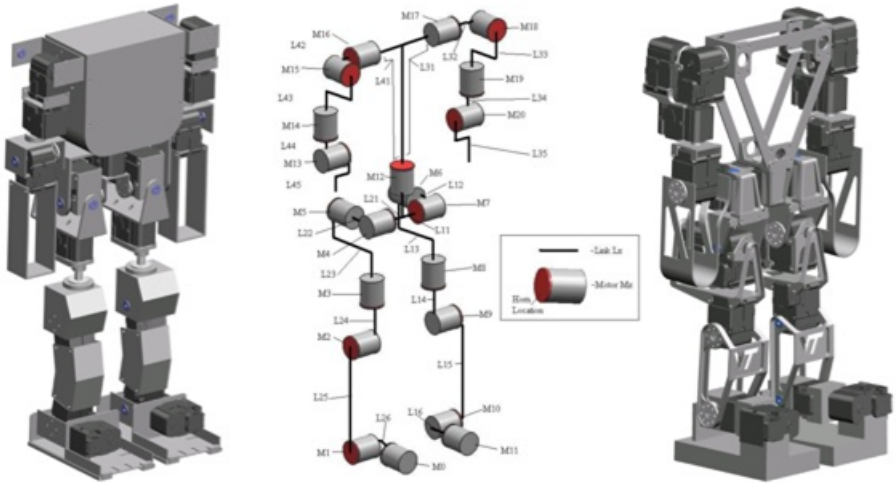


Figure 5.2: Kinematic and Mechanical Design of DARwIn

behavior are again required for successful operation. For RoboCup, a software module built using the general approach to behavioral programming presented in Chapter 3 is again used. By developing this implementation and comparing it with the Urban Challenge implementation, the portability of high-level behavioral programming across drastically different platforms and functionality requirements can be examined.

5.1 Robotic Platform

The Dynamic Anthropomorphic Robot with Intelligence (DARwIn) series of robots are a standardized humanoid robotics platform capable of bipedal walking and performing human-like motions. The robot stands 600 mm tall, weighs 4 Kg, and has 22[1] degrees-of-freedom (6 in each leg, 4 in each arm, one in the waist, and one in the neck). All joints are actuated by coreless DC motors that use distributed control with controllable compliance. DARwIn's computational power comes in the form of a PC-104+ computer stack with an Intel Pentium M 1.4 GHz processor mounted in the chest. Custom electronics provide power management to the computer, motors, and sensors from two 8.2V lithium polymer batteries attached to the legs. For sensing, DARwIn has two IEEE 1394 (Firewire) cameras mounted to the head and a 6 axis

[1] 20 DOF for the RoboCup competition

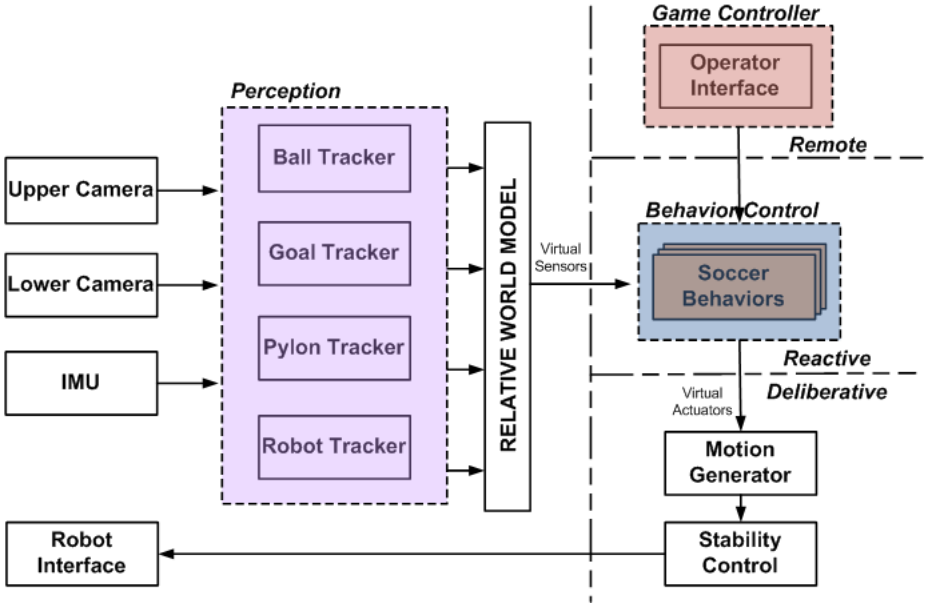


Figure 5.3: DARwIn's tri-level Hybrid control architecture

rate gyro/accelerometer (IMU). All the motors provide position feedback as well, giving DARwIn a sense of proprioception. All the links on DARwIn are custom machined and designed to be both strong and lightweight. External shock absorbers have also been installed to protect the robot from damage when falling. With this hardware, DARwIn can implement human-like walking gaits and perceive the world (like humans) through a set of "eyes" mounted in the head.

5.2 System Architecture

DARwIn utilizes a tri-level, Hybrid Deliberative/Reactive control architecture which can be seen in Figure 5.3. Unlike the VictorTango scheme, this architecture follows a *Remote-Reactive-Deliberative* progression. Essentially, a deliberative mission planner has been replaced with a human operated *Game Controller*. This Game Controller produces a pseudo mission plan that defines the robot position (attacker, defender, or goalie), attacking goal (blue or yellow), and opposing team color (Black, Magenta, or Cyan), at any given time. These settings are then used by the *Behavior Control* module to guide the selection of behaviors and resulting emergent

behavior. Asynchronous perception agents produce sensor independent perception messages in the form of virtual sensors which populate a world model. It is important to note that the world model used by DARwIn is not "global" and is instead "relative". With no localization software, all virtual sensors provide perception information relative to the robot's coordinate frame. This implies that the robot does not have explicit knowledge of where it is on the field. Without localization, Behavior Control is faced with a more difficult problem, especially with respect to positioning and team play. It is possible, however, to build Behavior Control in such a way that emergent soccer playing behavior results even without any form of localization. The virtual sensors available to Behavior Control are described here:

Goal Sensor The relative heading and distance of both the yellow goal and blue goal if currently in view. If either goal is not in view, the time since that goal was last seen is provided along with the last known relative heading and distance.

Ball Sensor The relative heading, distance and velocity of the ball. If the ball is not in view, the time since it was last seen is provided. Unlike the goal sensor, however, a prediction of the relative heading, distance, and velocity is given. This prediction is calculated using an internal model and the last known ball position and velocity.

Beacon Sensor The relative heading and distance to both the yellow-blue-yellow beacon and the blue-yellow-blue beacon if currently in view. These beacons are always located on the edge of the field at the halfway line, and could potentially be used for localization. If either beacon is not in view, the time since last seen is provided along with the last known relative heading and distance.

Obstacle Sensor The radius, relative heading, and relative distance to all obstacles currently in view. Obstacles are primarily composed of other robots but may include any other foreign object on the field. If the obstacle is another robot, the team identification is provided. If no obstacles are detected, this list is empty.

Closest Bounded Green Area A series of profile points defining the edge of the green area

that DARwIn is currently standing on. If DARwIn were standing directly in front of the goal, these points would define the inner boundary of the goal box.

With these virtual sensors and the high-level settings provided by the Game Controller as inputs, Behavior Control must produce the correct emergent behavior for playing soccer by issuing motion commands via virtual actuators. These virtual actuators dictate the physical actions of the robot with respect to how it is walking, where the head is looking, and whether or not a special action, such as a kick, dive or cheer, should be performed. From these virtual actuators a low-level motion controller determines what all 22 motor position should be. Unlike the Urban Challenge, however, this low-level motion control is not responsible for any type of obstacle avoidance. Instead, it is primarily concerned with dynamic stability, which is a much larger problem on DARwIn then on Odin. In fact, Motion Control has no concept of obstacles and utilizes only inertial data for "knee-jerk" reactions. It is also the responsibility of Motion Control to detect when the robot has fallen over and override all virtual actuator inputs until DARwIn has returned to a stable standing position. The virtual actuators which Behavior Control may command are described here:

Gait Motion A parametrized gait command dictating the type of walk to perform and how it should be performed. The available motion types are *forward, backward, side_left, side_right, turn_in_place,* and *stop.* Both *side_left* and *side_right* refer to a side-stepping motion. Depending on the motion type, certain parameters must also be provided. For all motion types except *turn_in_place*, a desired step size and desired curvature must be specified. Curvature is measured in centimeters and can be used to make the robot walk in an arc. There is a minimum curvature of 10 cm, so if Behavior Control would like to execute a turn with a smaller radius than this, it must specifically request a *turn_in_place* gait. For the *turn_in_place* motion type, a change in angle per step must be specified. As an example, the type of motion needed to orient the robot around a ball already at its feet is a side step with a small curvature.

Head Motion The desired heading of DARwIn's head with respect to the rest of its body.

Since the vertical field of view of the cameras cover from DARwIn's feet to above the horizon, only one degree of freedom is needed in the neck. This head position defines where the robot is looking horizontally and has a maximum angle defined by the rules that cannot be exceeded. This prevents the robot from being able to look directly backwards without rotating its body.

Special Action Any type of action which requires playback of a pre-recorded motion such as kicking, diving, or cheering. If a special action (SA) is requested, it will trump any other type of motion command, whereas a head motion and a gait motion could be executed simultaneously. Two categories of kicks are available, a straight kick and a side pass. Both kicks can be performed with either foot and with any kick a strength magnitude must be specified. Two types of dives are also available. One dive results in the robot fully extended on the ground, while the other dive keeps the robot in a stable position without falling down. A dive can be performed in either direction and the length of time to remain on the ground before standing back up must be specified. Finally, a SA for both cheering and being sad are also available.

5.3 Task Decomposition and Hierarchy

Unfortunately, due to time constraints, the Hierarchical State Machine presented in this section has not been fully developed and tested. The following discussion should therefore be viewed as a *suggested* implementation. As an exercise in behavioral decomposition, the RoboCup problem still presents some unique challenges not present in the Urban Challenge case study.

At the most abstract level, the behavior of individual soccer players is separated by team role, or position. For example, the behavior of a goalie should be significantly different then that of an attacker. Furthermore, the allowable actions of a goalie are different then those of an attacker. A goalie is allowed to dive and block the ball with its hands, whereas an attacker may only interact with the ball by kicking it. The highest level state machine within Behavior Control is therefore used to differentiate between *goalie, attacker,* and *defender* states. In

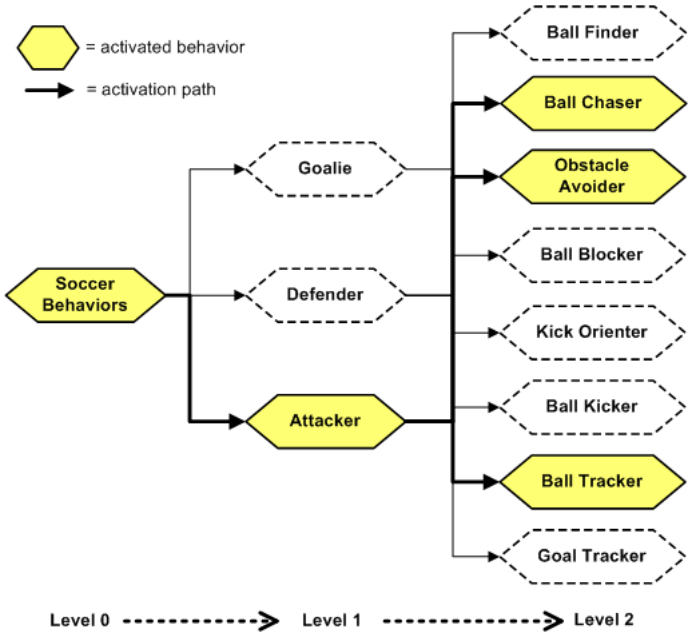


Figure 5.4: Behavior hierarchy suggested for robot soccer

each of these states a corresponding sub-behavior for each role is activated. The next level of behaviors are more skill oriented and essentially provide a library of options that can be called upon by either the `goalie`, `attacker`, or `defender` behaviors, as seen in Figure 5.4. Of these level 2 behaviors, we see behaviors differentiated mainly by a combination of what they are looking for and what type of action they hope to perform with respect to that object. For example, the `Ball Tracker` and `Goal Tracker` are interested in the ball and goal respectively, and are interested in tracking them using Head Motions. The `Ball Chaser`, `Ball Blocker`, and `Ball Kicker` are all interested in the ball, but with respect to approaching it by Gait Motions, blocking it through Special Action, or kicking it also by Special Action. In total, the right combination of level 2 behaviors should be sufficient for performing the following tasks: searching for the ball in an intelligent manner, approaching the ball in a manner that reduces the necessary amount of orienting towards the goal, approaching the ball as quickly as possible, orienting towards the opposing goal while keeping the ball at the robots feet, kicking the ball towards an opponents goal, clearing the ball away from our own goal as quickly as possible,

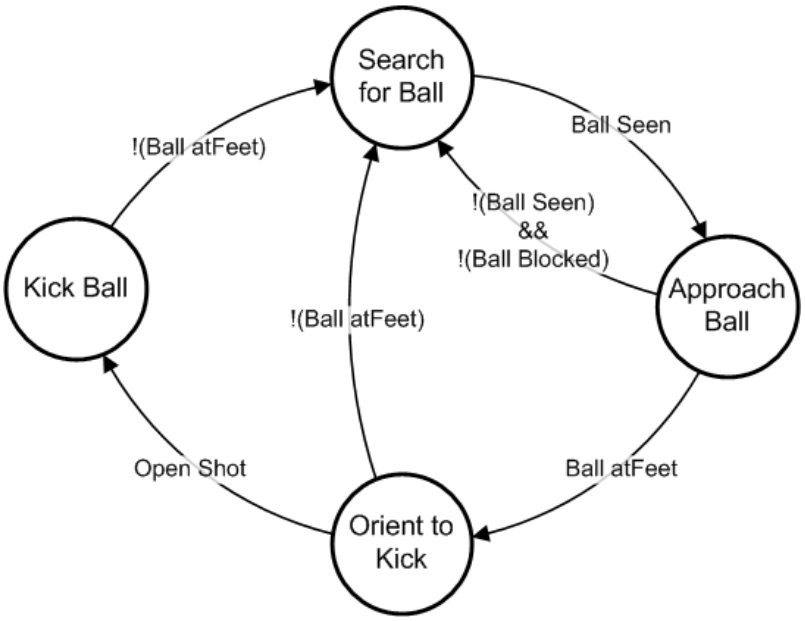


Figure 5.5: State diagram of the `Attacker`

maintaining a defensive position, and blocking a shot.

5.3.1 Attacker

Following the nomenclature set in Chapter 3, the `Attacker` is solely a *decision* behavior and therefore encodes no control policies. The main purpose of this behavior is to differentiate between more abstract situations and establish what sub-behaviors should be activated. This is done through the use of its internal states. A figure illustrating the state machine within the Attacker behavior is shown in Figure 5.5. In the *Search for Ball* state, the `Ball Finder` and `Obstacle Avoider` sub-behaviors are activated, resulting in the robot searching intelligently for the ball while avoiding any immediate obstacles. Once the ball has been seen, the `Attacker` enters the *Approach Ball* state, where the `Ball Chaser`, `Obstacle Avoider`, and `Ball Tracker` sub-behaviors are all activated. In this situation, the robot should move to the ball as quickly as possible while again avoiding obstacles. Now that the ball has been located, it is important not to lose sight of it until it is within range of the lower camera, so a separate behavior is used

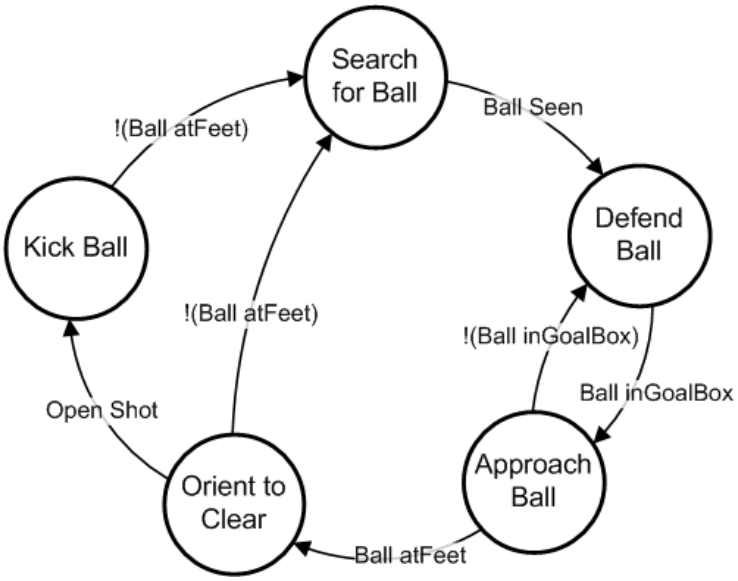


Figure 5.6: State Diagram of the `Goalie`

to keep track of the ball should it get kicked by another team. Once the robot has the ball at its feet, the `Attacker` transitions to the *Orient to Kick* state, where the `Kick Orienter` and `Goal Tracker` are activated. In this state, the robot has the ball at its feet so it searches for the goal using head motions and orients itself for the best shot on goal. Once an open shot has been evaluated, the *Kick Ball* state is entered which determines the most appropriate type of kick for the given situation via the `Ball Kicker` and `Goal Tracker` sub-behaviors.

5.3.2 Goalie

The Goalie, like the Attacker is a *decision* behavior, and not a *command* behavior. The internal states are fairly similar to the Attacker with the exception of a few key differences. The state machine diagram with transitions shown is seen in Figure 5.6. In the Goalie state machine, there is an additional *Defend Ball* state. In this state the activated sub-behaviors are the `Ball Blocker` and `Ball Tracker`. The `Ball Blocker` is responsible for staying in front of the ball, without approaching it, and in the case of a shot, selecting the appropriate dive to block it. Once the ball enters the goal box, however, the behavior becomes much more like a regular

field player. In this situation, the goalie should approach the ball, orient so as to clear the ball away from the goal, and select the most appropriate kick for doing so.

5.4 Command Fusion Mechanisms

While the majority of level 2 behaviors seen in Figure 5.4 have not yet been implemented and tested, their outputs in terms of control policies are already known. No matter what the situation is, at least one control policy with respect to each virtual actuator, π_{gait}, π_{head}, and π_{SA} must be active. It is therefore still possible to look at useful ways of resolving conflicts by the selection of command fusion mechanisms.

Similar to situations seen in the Urban Challenge, the Special Action virtual actuator has a finite set of possibilities. It therefore makes the most sense to use an urgency based voting mechanism similar to what was used on Odin. Using such a method is also important because Special Actions override all other motion types and take complete control of the robot. All behaviors that maintain a Special Action control policy must therefore incorporate an urgency policy as well.

Both the Head Motion and Gait Motion virtual actuators pose a very different problem, however. A potentially infinite range of both Head and Gait Motions exist, allowing for the possibility of greater *cooperation* amongst competing behaviors. For example, it is usually desirable to both avoid an obstacle and proceed towards a goal at the *same time.* This is a well known problem for behavior-based obstacle avoidance and methods such as potential fields [13] and motor schemas [4] have long since been used. These mechanisms work by finding a weighted average of all competing behavior's desires and executing this average. When the desires of competing behaviors are in the same general ballpark, such mechanisms usually prove sufficient. However, when behavioral outputs are completely contradictory, average-based superposition mechanisms can result in action that is beneficial to neither behavior. For example, it would be undesirable if in an attempt to track both the goal and the ball, the robot were to look directly in between the two and subsequently lose sight of both.

Another, potentially more robust command fusion ASM takes advantage of multi-valued behavior outputs. Behaviors typically produce a single-valued output that from their perspective is the optimal action to perform. Using the aforementioned fusion ASM, the output with the highest urgency is then selected. Suppose now that each behavior produced a multi-valued output such that the behavior would be satisfied with all values. This range, or subset, of values could be ranked by preference, and when compared to another behavior's outputs, a final selection can be made that satisfies both behaviors. This allows for the ability to trade-off the benefits of different behaviors. Examples of utilizing multi-valued behavior outputs has been performed by Rosenblatt and Payton in [24]. Of course, such a fusion ASM would still require tweaking by the designer to find the appropriate degree to which the mediocre satisfaction of multiple behaviors outweighs the complete satisfaction of one behavior. A cooperative fusion mechanism such as this could be useful for balancing the desire to use a Head Motion to track both the ball and the goal. Should no solution exist that satisfies both behaviors, it is still possible to choose an action which best satisfies one behavior.

5.5 Performance Results

As with the Urban Challenge, exhaustive testing in simulation and on the actual robot are paramount to the overall performance. For this purpose, a simulation environment developed at the Technische Universitat Darmstadt (TUD) has been taken advantage of. The Multi-Robot-Simulation-Framework (MuRoSimF) [10] provides real-time kinematic motion simulation, collision detection, and sensor simulation for multiple robots at once. Interfacing DARwIn's complete control and perception architecture with MuRoSimF allows for behavioral testing on an individual and team level. With accurate camera simulation, it is also possible to test the behavioral robustness to realistic sensor input. A screen shot of the simulation environment is seen in Figure 5.7.

Through simulation and live testing, controlled, repeatable testing can be performed to evaluate the action selection process on DARwIn. Realistic game scenarios can be created and

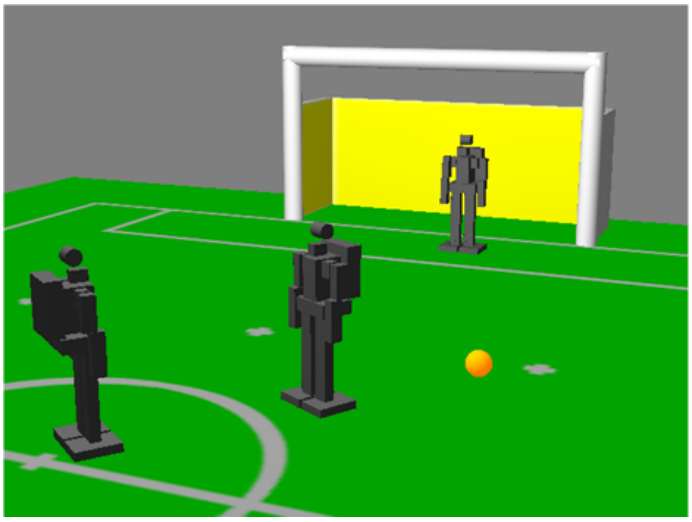


Figure 5.7: MuRoSimF screen shot used for testing and evaluation

the overall performance can be examined. With the ability to run multiple robots at once, it is also possible to investigate the effect of small behavioral changes over the course of an entire game. Unlike the urban driving scenario, it is not wholly sufficient to simply follow the rules of the road. Due to the unstructured nature of soccer, small differences in behavior can have great effect on the outcome of the game. Evaluating the performance of higher-level strategy will therefore be imperative to being competitive at competition.

5.6 Lessons Learned

The work performed so far on a behavioral module for the RoboCup problem illustrates the potential portability of the general approach to behavioral programming presented in Chapter 3. A formulation of the problem in terms consistent with the Urban Challenge case study is clearly possible. While the inputs, as virtual sensors, outputs, as virtual actuators, and behavioral requirements are completely different, the same methodology for behavioral coordination can clearly be used. The details of the hierarchical behavior tree and the internal state machines within those behaviors may not yet be solved for, but with a capable simulator and testing

plan, the solution is within reach.

The unique virtual actuators on DARwIn also allow for more intelligent command fusion mechanisms not used on Odin. Specifically, superposition, fuzzy, and multiple-value mechanisms can be integrated, reducing the reliance on the designer to be aware of *all* behavioral interactions. These methods also make it possible to add important responsibilities to the behavioral component that may otherwise be handled elsewhere in the Hybrid architecture, such as obstacle avoidance and directed sensor focus. While performing behavior-based obstacle avoidance is clearly possible, it is important to remember that doing so comes with the shortcomings of any reactive paradigm, such as the lack of optimality and performance guarantees.

Finally, the RoboCup formulation shows some of the portability of low-level behaviors within the hierarchical tree. In this implementation, we see the reuse of level 2 behaviors such as the `Ball Chaser` by all three level 1 behaviors, `Goalie, Attacker` and `Defender`. The lower-level behavior can remain active through transitions between higher-level behaviors and there should be no interruption in the execution of the lower behavior. Looking at these situations has provided further insight into the hierarchical relationship between behaviors in the case of commonality. In essence, if a behavior should be activated with multiple other behaviors, but only when those behaviors are in certain states, then it should be placed lower in the hierarchy. If a behavior should be activated during all states of only one other behavior, then they should be in parallel. Finally, if a behavior should be activated during all states of multiple behaviors all on the same level, then that behavior should be higher in the hierarchy.

Chapter 6

Conclusions, Contributions, Observations, and Future Work

The potential uses for mobile robots in industry, the home, the military, and entertainment are continually growing. Robots capable of performing dull, dirty, and dangerous jobs have the ability to save countless human lives and improve people's everyday quality of life. Before this is possible, however, significant advances must be made to bring robots out of the lab and into the field.

6.1 Summary of Contributions

Research in mobile robotic control architectures over the past 30 years have brought robots significantly closer to this reality. From the Hierarchical Paradigm to the Reactive Paradigm, we have seen robots gain the ability to tackle both small and large technical challenges, mimicking biological intelligence in some aspects and taking advantage of brute force computing in others. Uniting these abilities in a coherent, fully autonomous system capable of solving complex, sophisticated tasks remains the challenge for most researchers today. Hybrid Deliberative/Reactive approaches hope to bridge this gap, but the division of responsibilities within such architectures remain in question. Most traditional Hybrid architectures take a bi-level approach, with deliberative components responsible for advanced cognitive functions placed at a higher level, and reactive, behavior-based components responsible for direct actuator con-

trol placed at a lower level. With the rapid growth of computing technology, however, there has been a re-emergence of deliberative methods for low-level motion planning. Such methods provide the important traits of predictability and optimality, which are extremely useful from an engineering point of view. This trend, along with the need for robots capable of handling more and more complex problems, has resulted in the growth of tri-level Hybrid architectures. Such architectures exhibit a *deliberative-reactive-deliberative* progression, thereby changing the typical scope and application of behavior-based software agents.

Behavioral control components now have the responsibility of bridging the gap between high-level mission planning and low-level motion control. Within the tri-level Hybrid architecture, such a component receives perception information through *virtual sensors* and dictates desired action through *virtual actuators.* This behavioral component requires the ability to break down complex problems into smaller sub-goals and tasks whose importance may change dynamically depending on the robot's current situation. Based on this *contextual intelligence*, the right selection of behaviors must be made, whose outputs can then be combined to produce the proper *emergent behavior* for the task at hand. These two important properties of biological intelligence are encompassed in the *action selection problem.* Traditional action selection mechanisms have not typically been used for such higher-level problem solving, and so most mechanisms are not individually sufficient. It is possible, however to combine existing approaches to unite their individual advantages and reduce the effect of their individual drawbacks. In this thesis, a novel method of placing an arbitration ASM *in sequence* with a command fusion ASM has been presented. Specifically a Hierarchical State Machine is used for behavioral selection, followed by an application-specific command fusion mechanism for handling behavioral conflict.

This approach to behavioral programming and action selection has been validated in the DARPA Urban Challenge, a landmark robotics problem. In this event, Odin, team Victor-Tango's entry was able to negotiate difficult scenarios involved with driving a large unmanned ground vehicle through intersections, parking lots, and multi-lane roads, all in the presence of live traffic. Placing third overall, Odin's behavioral implementation proved flawless throughout the course of the race. The general approach to behavioral programming presented in this

thesis is also being applied to the RoboCup soccer competition, another landmark challenge. Through this new implementation, we see the versatility and portability of such an approach despite drastically different base platforms, virtual sensors, virtual actuators, and behavioral requirements. Together, these two important case studies have exposed some important observations which are summarized here.

6.2 Important Observations

6.2.1 Hierarchy

By using hierarchy for behavior decomposition, there exists an intuitive method for the robot designer to breakdown tasks according to their level of abstraction. This top-down approach is a natural thought process which can be applied to a variety of problems and scenarios. When implemented properly, such a behavioral tree provides the robot with adequate *situatedness*. The proper use of hierarchy also encourages the reuse of lower-level behaviors in different situations and supports ideas behind object-oriented programming. For example, behavioral commonalities should not be reproduced multiple times and can instead be simply placed higher in the hierarchy. Finally, using a hierarchical breakdown as opposed to a completely flat structure prevents massive growth in computational requirements as the number of behaviors in a system grows.

6.2.2 State Machines

By utilizing state machines within each behavior, the robot designer is again provided with important behavioral tools. State machines allow for the definition of *temporal sequencing* when necessary. The designer can use state transitions to easily differentiate when order is important and when it is not with respect to the completion of certain tasks. State transitions are also useful from a systems engineering point of view by predefining the perception requirements of the robot. Furthermore, the correct design of state transitions provides a simple mechanism

for handling perception noise. Being robust to perception error is not only essential, but particularly useful with respect to *goal-orientedness* and *persistence*.

6.2.3 Command Fusion

Through the use of multiple command fusion mechanisms organized by virtual actuator, flexibility is given to the overall action selection system. How behavioral conflict is resolved has the greatest effect on the *emergent behavior* of the robot. Constricting the designer to only one method is detrimental when certain methods work better for certain applications and certain virtual actuators.

6.3 Future Work

While this approach to behavioral programming provides the designer with a useful tool set, the final performance is still completely reliant on how well these tools are used. There is no "silver bullet" solution that will immediately produce intelligent behavior. Exhaustive testing both in simulation and on the actual robot is imperative to success. Because of this shortcoming, two major areas for future research should be addressed. First, research into better methods for generation of the hierarchical state machine should be performed. Being able to formulate a behavioral problem in a standard way such that construction of the HSM is simplified, would be extremely beneficial. It would result in more consistent performance amongst different implementations and make the job of the robot designer significantly easier. Establishing such a method would first require a deeper analysis of behavioral HSMs. The effect of tree size and shape on overall performance must be analyzed, and a greater understanding of state reachability, structural succinctness, and equivalence is needed. Some research in this area with respect to general HSMs can be found in [2] and should be applied to the behavioral variation presented here.

The other major area for future research involves the use of machine learning and optimization techniques to tweak control policies, state transitions, and command fusion parameters.

Formalizing such an approach to work across all behaviors poses some very difficult problems. While simulation would still be required for extensive training, the need for supervision and intervention of a human overseer would at least be removed. The reliability of training a system in simulation, however, and then moving it to the real world poses further problems of its own. Either way, a behavioral structure capable of learning from experience would be paramount as robots are expected to complete more and more complex tasks with situations that a designer could not possibly predict.

Bibliography

[1] Albus, J. S. (1991). "Outline for a Theory of Intelligence." In *IEEE Trans. on Systems, Man, and Cybernetics,* Vol. 21, No. 3, May/June.

[2] Alur, R., Yannakakis, M. (2001). "Model Checking of Hierarchical State Machines." *ACM Transactions on Programming Languages and Systems,* Vol. 23(3): 273-303.

[3] Argall, B., Browning, B., Veloso, M. (2007). "Learning to Select State Machines using Expert Advice on an Autonomous Robot." In *IEEE International Conference on Robotics and Automation,* pages 2124-2129.

[4] Arkin, R. C. (1987). "Motor schema based navigation for a mobile robot: An approach to programming by behavior." In *IEEE International Conference on Robotics and Automation,* pages 264-271.

[5] Arkin, R. C., Riseman, E. M., and Hansen, A. (1987). "AuRA: An Architecture for Vision-Based Robot Navigation," proceedings of the *DARPA Image Understanding Workshop*, pp. 413-417. Los Angeles, CA.

[6] Brooks, R. A. (1986). "A Robust Layered Control System for a Mobile Robot." In *IEEE Journal of Robotics and Automation,* Vol. 2 (1): 14-23.

[7] Bryson, J. (2000). "Hierarchy and Sequence vs. Full Parallelism in Action Selection." In J.A. Meyer, A. Berthoz, D. Floreano, H. Roitblat, and S.W. Wilson, eds., *Proc. Sixth Intl. Conf. on Simulation of Adaptive Behavior,* 147-156. Cambridge, MA: MIT Press.

[8] DARPA. (2007). "Urban Challenge Technical Evaluation Criteria." http://www.darpa.mil/grandchallenge/rules.asp. Defense Advanced Research Projects Agency, Arlington, VA.

[9] Dechter, R., Pearl, J. (1985). "Generalized best-first search strategies and the optimality of A*." In *Journal of the ACM,* Vol. 32(3): 505-536.

[10] Friedmann, M., Perterson, K., Stryk, O. v. (2007). "Adequate Motion Simulation and Collision Detection for Soccer Playing Humanoid Robots" In *Proc. 2nd Workshop on Humanoid Soccer Robots at the 2007 IEEE-RAS Int. Conf. On Humanoid Robots.*

[11] Gat, E., (1998). "Three-layer Architectures," *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonasson, R. Murphy, editors, Cambridge, MA: MIT Press.

[12] Harel, D. (1987). "Statecharts: A Visual Formalism for Complex Systems." In *Science of Computer Programming,* Vol. 8: 231-274.

[13] Khatib, O. (1986). "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots." *The International Journal of Robotics Research,* 5(1):90 - 98.

[14] Konolige, K., and Myers, K. (1998). "The Saphira Architecture for Autonomous Mobile Robots," *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonasson, R. Murphy, editors, Cambridge, MA: MIT Press.

[15] Kristensen, S. (1996). *Sensor Planning with Bayesian Decision Analysis.* PhD thesis, Department of Medical Informatics and Image Analysis, Aalborg University.

[16] Mackenzie, D., Arkin, R., and Cameron, J. (1997). "Specification and Execution of Multiagent Missions." *Autonomous Robots*, 4(1).

[17] Maes, P. (1989). "How To Do the Right Thing." *Technical Report NE-43-836*, AI Laboratory, MIT. Cambridge, MA.

[18] Minsky, M. (1985). *The Society of Mind.* Simon and Schuster, New York, NY.

[19] Muecke, K., and Hong, D. W. (2008). "The Synergistic Combination of Research, Education, and International Robot Competitions Through the Development of a Humanoid Robot." *32nd ASME Mechanisms and Robotics Conference*, New York City, NY.

[20] Muecke, K., and Hong, D. W. (2007). "DARwIn's Evolution: Development of a Humanoid Robot." *IEEE International Conference on Intelligent Robotics and Systems.* October 29 - November 2, 2007.

[21] Murphy, R. R. (2000). *Introduction to AI Robotics.* Cambridge, MA: MIT Press.

[22] Murphy, R., and Mali, A. (1997). "Lessons Learned in Integrating Sensing into Autonomous Mobile Robot Architectures." *Journal of Experimental and Theoretical Artificial Intelligence special issue on Software Architectures for Hardware Agents,* vol. 9, no. 2, pp. 191-209.

[23] Pirjanian, P. (1999). "Behavior Coordination Mechanisms - State-of-the-Art." *Tech Report IRIS-99-375*, Institute for Robotics and Intelligent Systems, University of Southern California, Los Angeles, California.

[24] Rosenblatt, J. (1995). "DAMN: A Distributed Architecture for Mobile Navigation." In *AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, *Stanford, CA*. AAAI Press, Menlo Park, CA.

[25] Russel, S., and Norvig, P. (2003). *Artificial Intelligence - A Modern Approach.* Upper Saddle River, New Jersey, Pearson Education, Inc.

[26] Saffiotti, A. (1997). "The Uses of Fuzzy Logic in Autonomous Robot Navigation: a catalogue raisonne." *Technical Report 2.1,* IRIDA, Universite Libre de Bruxelles, 50 av. F. Roosevelt, CP 194/6, B-1050 Brussels, Belgium.

[27] Simmons, R., Goodwin, R., Haigh, K., Koenig, S., and O'Sullivan, J. (1997). "A Layered Architecture for Office Delivery Robots," proceedings *Autonomous Agents 97,* pp. 245-252.

[28] Simon, H. A. (1960). *The New Science of Management Decision.* Harper and Row, New York.

[29] Thrun, S., Montemerlo, M, et. Al. (2006). "Stanley: The robot that won the DARPA Grand Challenge: Research Articles," *Journal of Field Robotics,* vol. 23, no. 9, September, 2006, pp. 661-692.

[30] Tyrell, T. (1993). *Computational Mechanisms for Action Selection.* PhD thesis, University of Edinburgh.

[31] Urmson, C., et. Al. (2006). "A Robust Approach to High-Speed Navigation for Unrehearsed Desert Terrain." *Journal of Field Robotics*, vol. 23, no. 8, August, 2006, pp. 467.

[32] VictorTango et Al. (2008). "Odin: Team VictorTango's Entry in the DARPA Urban Challenge." In *Journal of Field Robotics - Special Edition on the DARPA Urban Challenge* (submitted).

[33] Yen, J., and Pfluger, N. (1995). "A Fuzzy Logic Based Extension to Payton and Rosenblatt's Command Fusion Method for Mobile Robot Navigation." *IEEE Transactions on Systems, Man, and Cybernetics,* 25(6):971 - 978.

Appendix A

Behavioral State Diagrams

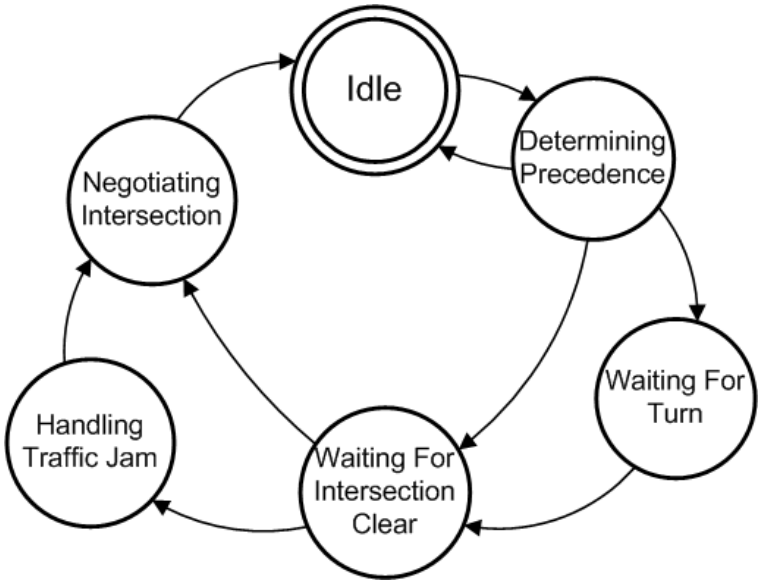


Figure A.1: State Diagram of the `Precedence Driver`

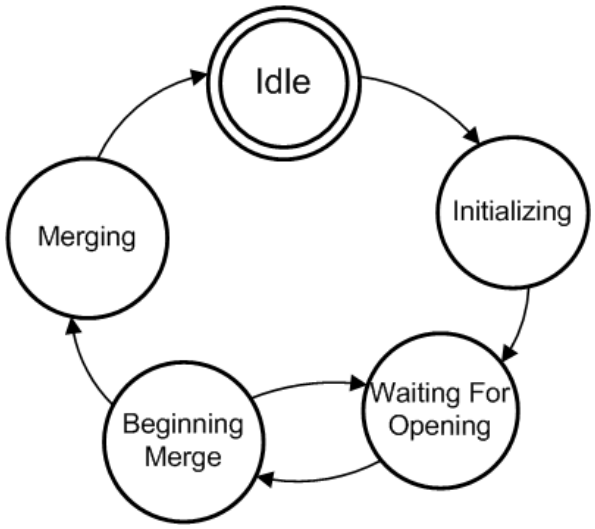


Figure A.2: State Diagram of the `Merge Driver`

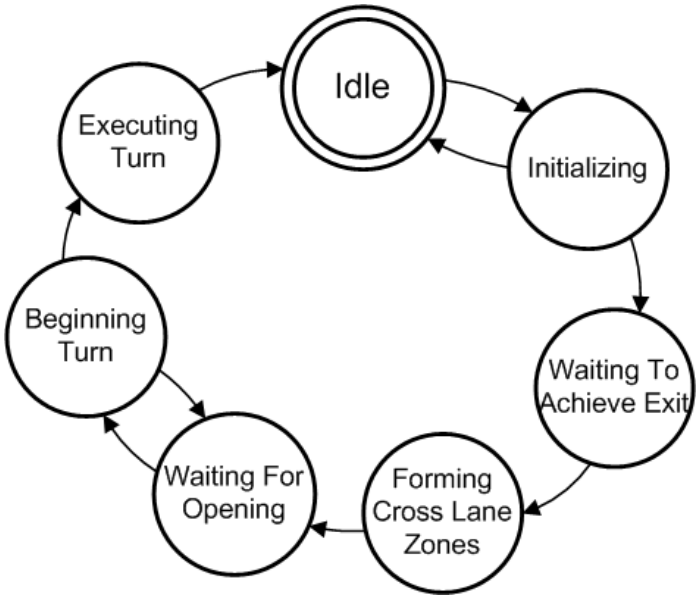


Figure A.3: State Diagram of the `Left Turn Driver`

Appendix B

Detailed System Architectures

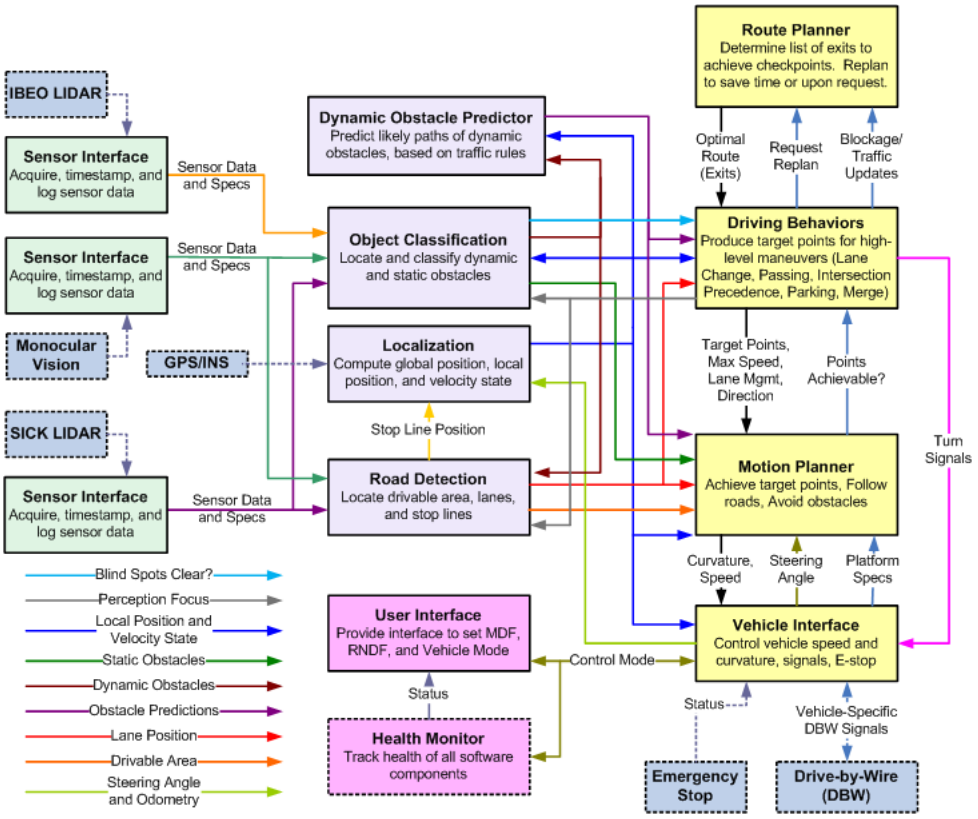


Figure B.1: System Architecture used for the DARPA Urban Challenge

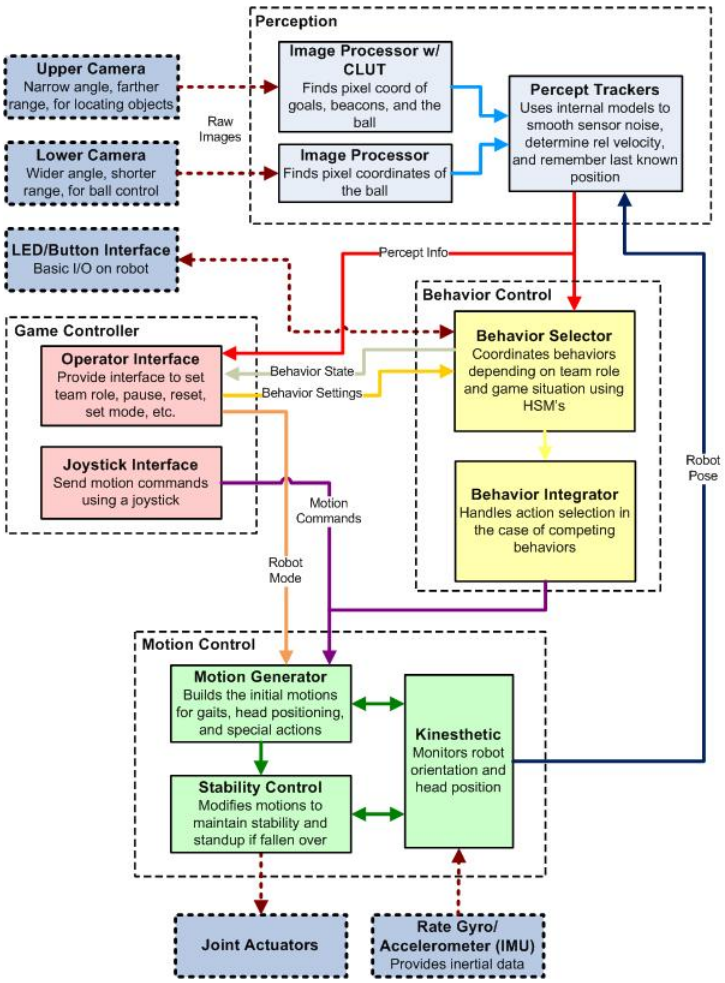


Figure B.2: System Architecture used for RoboCup