JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 5, December 2008

VictorTango Architecture for Autonomous Navigation in the DARPA Urban Challenge

Jesse Hurdus, Andrew Bacha, Cheryl Bauman, Stephen Cacciola, Ruel Faruque, Peter King, and Chris Terwelp[*]
TORC Technologies, Blacksburg, VA, 24060

Patrick Currier, Dennis Hong, and Alfred Wicks[†]
Virginia Tech, Blacksburg, VA, 24060

and

Charles Reinholtz[‡]
Embry-Riddle Aeronautical University, Daytona Beach, FL, 32114

DOI: 10.2514/1.40547

To solve the autonomous urban driving problem presented by the 2007 Defense Advanced Research Project Agency Urban Challenge, team VictorTango developed a tri-level architecture with a deliberative-reactive-deliberative structure. The VictorTango architecture emphasizes a robust, reusable, and modular design, using hardware independent *virtual actuators* and *virtual sensors*. Details of the Urban Challenge implementation are provided, highlighting the functionalities and interactions of different modules within the architecture. Situational examples from the Urban Challenge problem illustrate the performance of the architecture in real-world situations, and communications latencies with their effect on decision making are analyzed. Finally, recommendations for future refinement of the architecture are presented. The VictorTango architecture ultimately proved capable of completing the Urban Challenge problem. Furthermore, the modularity, hardware independence, and robustness of the architecture have enabled it to be applied to other platforms and other problems in the Unmanned Systems field.

I. Introduction and Overview

A. Paper Overview

MOBILE robotics applications are pushing the envelope for autonomous vehicle technology in many new directions. Problems such as robot cognition, sensor fusion, behavior coordination, and path planning are being brought out of the lab and into the field [1,2]. The Defence Advanced Research Projects Agency (DARPA) Urban Challenge is a landmark technical problem that requires bringing effective techniques from all these areas together into a complete system. To succeed in the Urban Challenge, individual solutions and algorithms must complete their responsibilities efficiently and effectively, but even more important these solutions must act coherently in a greater system. The entire autonomous system must be reliable enough to operate for 6 h without human intervention, robust

Received 22 August 2008; accepted for publication 3 September 2008.

[*] 2200 Kraft Drive, Suite 1325, www.torctech.com

[†] Department of Mechanical Engineering, Virginia Tech (0238), victortango@vt.edu

[‡] Chair, Department of Mechanical Engineering, 600 South Clyde Morris Boulevard

enough to handle failures, fast enough to react to surprises at 30 mph, and intelligent enough to tackle the complex problem of driving in an urban environment.

Completing such sophisticated tasks in a dynamic, partially observable, and unpredictable environment requires a flexible system architecture that balances both *planning* and *reactivity*. To accomplish this, team VictorTango, a partnership between Virginia Tech and TORC Technologies LLC, has developed a tri-level, Hybrid Deliberative/Reactive architecture. This architecture takes advantage of optimal, search-based, planning techniques as well as reactive, behavior-based methods. It follows a *deliberative–reactive–deliberative* progression that sandwiches a behavioral component between a high-level mission planner and a low-level motion planner. World modeling is handled in parallel, providing sensor-independent perception messages to the control modules by way of *virtual sensors*. Hardware independence is then maintained through the use of *virtual actuators*. All software agents operate asynchronously and communicate according to the Joint Architecture for Unmanned Systems (JAUS) protocol, an emerging industry standard. As a result, the VictorTango software architecture is very modular in design, with the added benefit of being portable and scalable. Health monitoring of both hardware and software is continuously performed, at different levels, allowing for various error recovery techniques that depend on the abstraction level of the problem. Lastly, this architecture is well-suited for rapid deployment by a team of developers in a situation that requires an immense amount of testing and design cycles, in both simulation and on the vehicle.

In this paper, the unique problem posed by the DARPA Urban Challenge and the hardware and sensing platform used by team VictorTango is presented. A brief history of autonomous mobile robot software architectures illustrates the progression of the field, and provides a context for discussing the improvements made with the VictorTango Architecture. Next, a detailed discussion of the implementation is given, focusing on the distribution of responsibilities and the balance between deliberative and behavior-based approaches. Finally, results are given to exemplify the interaction of software modules within the system. The flow of information and the chain of decision making are presented in several different examples and an analysis of communication latency illustrates the effect on overall reaction time.

B. Problem Definition

The DARPA Urban Challenge required an autonomous ground vehicle to navigate an ordered list of checkpoints in a road network. The road network is supplied as a priori information in the form of a route network definition file (RNDF) that lists each drivable road as well as regions called zones, used to represent parking lots or other unstructured environments. The RNDF describes each lane by a series of waypoints with certain waypoints flagged as entrances and exits that connect each lane to the road network. A series of mission data files (MDFs) are provided that specify checkpoints which must be visited in a specified order. To achieve the MDF checkpoints as fast as possible, the vehicle must choose roads considering road speed limits, possible road blockages and traffic conditions. The vehicle must obey California State driving rules while driving safely and defensively, avoiding both static obstacles and moving traffic at speeds of up to 30 mph. Finally, the autonomous vehicle must be able to complete the 60 mile competition in less than 6 h.

C. Hardware Overview

The base platform is a 2005 Ford Escape Hybrid converted by the team to meet the DARPA requirement of an autonomous full-size commercial vehicle platform. The Escape, dubbed Odin, is outfitted with a highly integrated custom drive-by-wire system and is shown in Fig. 1. The rear cargo area holds the main computing system, sensor communication modules, and power distribution. Power for the autonomous systems is drawn from the hybrid battery pack and computing is supplied by two dual-quad core servers.

Odin's exteroceptive sensor suite (Fig. 2) covers the full 360° area around Odin. Primary obstacle detection is provided by three IBEO light detection and ranging (LIDAR) units. Two IBEO XT ALASCA LIDARs are mounted at the front corners of Odin, covering a combined 260° horizontal field of view, as shown in Fig. 3. These two sensors operate as a coordinated pair through IBEO's Fusion system. The third IBEO unit, an ALASCA A0, is mounted on the rear bumper and covers a 150° horizontal field of view. All three sensors contain four scan planes that span a maximum of 3.2 vertical degrees. The IBEO's are capable of processing multiple returns for each laser pulse, effectively allowing them to see through rain and light dust clouds.

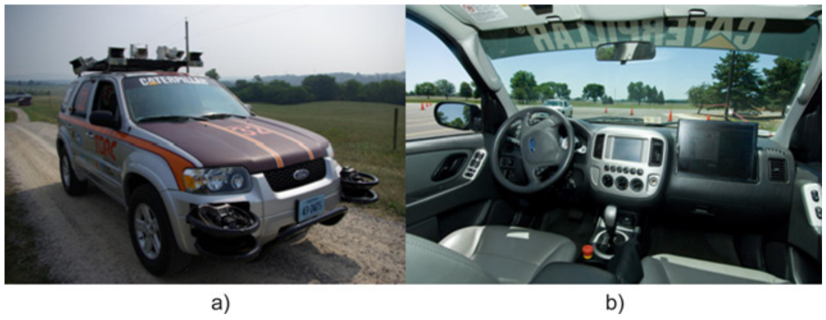

Fig. 1 a) Odin undergoing testing at Virginia Tech, b) Front seat view of Odin.

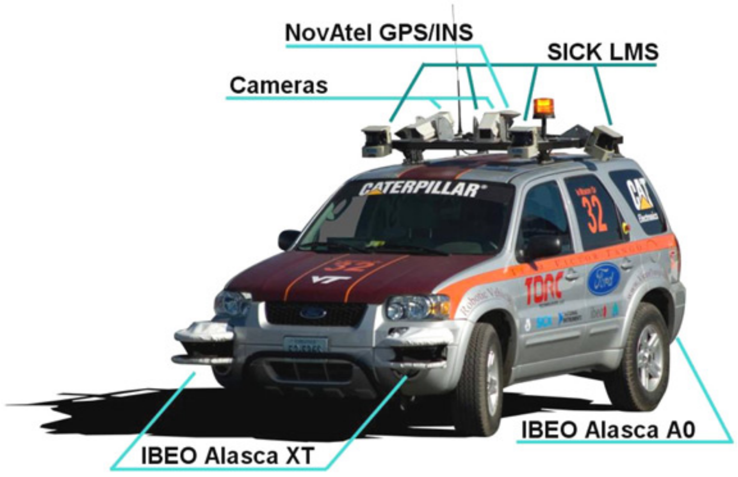

Fig. 2 Odin's sensor layout with sensors labeled.

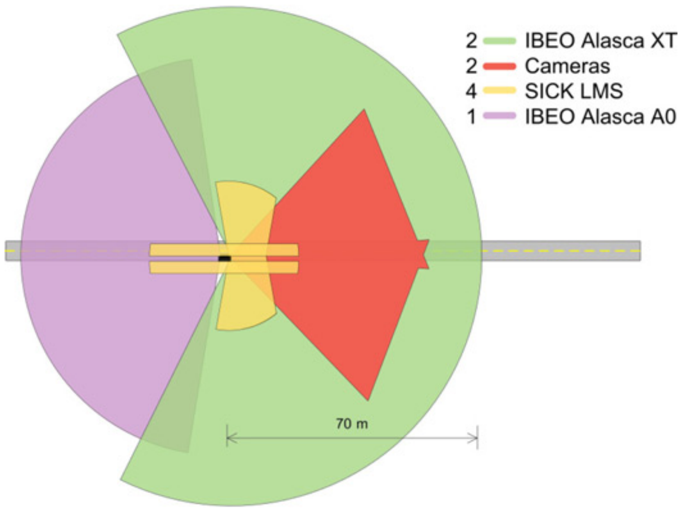

Fig. 3 Coverage provided by the sensors mounted on Odin (the black rectangle in the center is Odin).

Four SICK LMS 291 LIDAR units used to supplement the IBEOs are mounted on the roof rack. Two SICKs point forward and down at angles of 8 and 10° below horizontal, resulting in an approximate range on level ground of 12 and 10 m respectively. These units were used to supplement the IBEO's object detection capabilities by detecting small objects, ditches, and curbs. The other two SICKs are mounted on each side of the vehicle, and are used to monitor Odin's blind spot during lane changes.

Additional sensors include two IEEE 1394 color cameras mounted on the roof rack facing forward, covering a 90° horizontal field of view. These cameras are capable of transmitting raw 1024 by 768 images at up to 15 frames per second, and are used to detect roads and assist in object classification. A NovAtel Propak LB-plus GPS/INS unit is used to provide a commercial off-the-shelf (COTS) filtered positioning solution. Coupled with an OmniSTAR HP subscription, the unit has a rated circular error probable (CEP) of 10 cm.

D. Review of Software Architectures

The origins of mobile robot control are rooted in the Hierarchical Paradigm, as defined in [3]. The emphasis on planning in this approach pushed the development of more effective search algorithms, such as Dijkstra and A*. These algorithms allow an agent to find an optimal "sequence of actions that achieves its goals, when no single action will do" [4]. 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 cluttered room. As the number of objects being represented in the world increases and the number of possible actions broadens, the search space grows exponentially. Subsequently, the planning component became a bottleneck.

In response to these shortcomings, the reactive paradigm incorporates important ideas from *ethology*: the study of animal and insect behavior. The main feature of the reactive paradigm is that all actions are accomplished through the use of distinct behaviors. The overall architecture is 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 [5]. Essentially, planning is thrown out in favor of simpler, more efficient, reactive behaviors. The result is a parallel, vertical decomposition of behaviors that is much faster in operation, requires less computational power, and is more robust to failure. However, it is also much more imprecise. It is difficult to determine beforehand exactly what discrete behaviors are needed to combine and produce a rich emergent behavior. Even when the correct base behaviors are determined, exactly how they are combined has a huge effect on what emergent behavior results. This is known as the action selection problem. Furthermore, without any sort of planning component, purely reactive architectures eliminated any form of memory or reasoning about the global state of the robot. 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.

Owing to the shortcomings of 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 subgoals can be achieved are necessary for more complex problems. The result is the bi-level Hybrid Deliberative/Reactive Paradigm, which presents the idea of combining reactive behaviors with top-down, hierarchical planners used for more deliberative functions. These deliberative components are designed to run asynchronously, providing a set of intermediary goals for use as guidance to a lower level reactive system. These intermediary goals are intended to be sufficient for preventing the reactive system from making too many poor decisions as well as provide the ability to recover from being "trapped".

A loose analogy of hybrid architectures can be made to the way the 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. These software modules are free to use whichever paradigm is most appropriate for their individual task, whether it be search-based or behavior-based. This allows for a deeper abstraction of responsibilities within the overall control architecture. Such an approach also lends itself well to object-oriented programming and the portability of specialized modules to new domains. The question then becomes how to organize these software modules and determine what information should be passed between them. How does the overall behavior emerge? When properly implemented, a hybrid approach balances reactivity to unforeseen events with the execution of

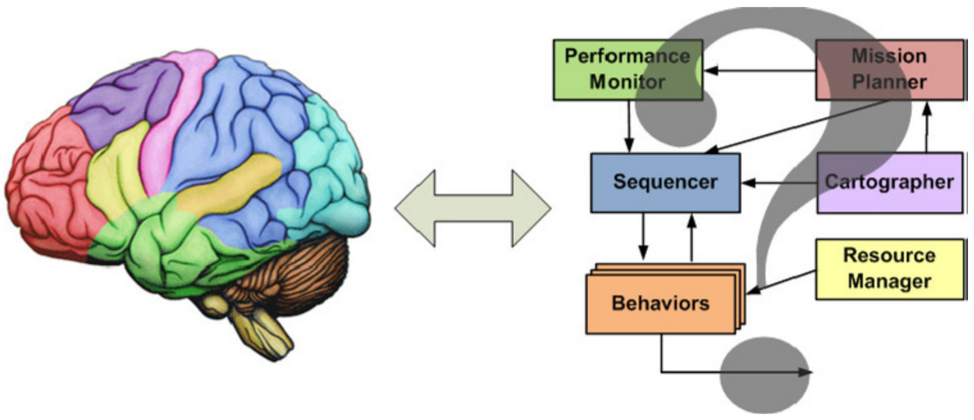

Fig. 4 Human brain/hybrid architecture analogy.

planned actions, similar to what is seen in biological intelligence. Figure 4 illustrates this problem and analogy to the human brain.

The question remains as to where to draw the line between planning components and reactive components. The early success of well-known Reactive, or Behavior-Based architectures such as Subsumption Architecture [5] or Potential Fields [6] in the mobile robot navigation domain were due to their speed, elegance, and simplicity. Reactive architectures subsequently became accepted as the “best” way of doing low-level motion control [3]. 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 are done on the higher, deliberative layer, while selected behaviors run 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) [7], Sensor Fusion Effects (SFX) [8], 3T [9], Task Control Architecture (TCA) [10], and Saphira [11]. The Saphira architecture, developed at the Stanford Research Institute (SRI) for use on the Pioneer robot platform is a good example of the *bi-level* hybrid approach and is seen in Fig. 5.

The deliberative layer is needed to bring robotic applications into more cognitively challenging domains beyond just navigation. The reactive layer is then responsible for more time critical control problems such as low-level obstacle avoidance, allowing the robot to react to unforeseen events and surprises. Because of the nature of the reactive paradigm, however, there is no guarantee of optimality beyond the intermediary goals set by a deliberative planner. Even if the robot goes in the right direction, it could easily spend extra time getting into and out of box canyons or other situations where behavior-based navigation algorithms have problems. Fortunately, with the rapid growth of computing technology, there has been a re-emergence of deliberative methods for low-level motion planning [2,12]. 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. A new type of hybrid deliberative–reactive architecture is needed to take advantage of these methods, while maintaining the benefits of behavior-based systems.

II. VictorTango’s Tri-level Architecture

This section describes the VictorTango Architecture, a tri-level hybrid deliberative/reactive software architecture. The system-level features of this architecture are highlighted and the organization of perceptive, deliberative, and reactive software components is shown. A detailed discussion of individual components illustrates the distribution of responsibilities within the architecture. A description of the system-level Health Monitor is then given, followed with an overview of the communications backbone that enables the portability and scalability of the architecture.

A. Architecture Overview

Traditionally, Hybrid architectures utilize a bi-level layout, with only one division between deliberative and behavior-based components. This results in robust low-level motion control, but lacks the optimality and repeatability

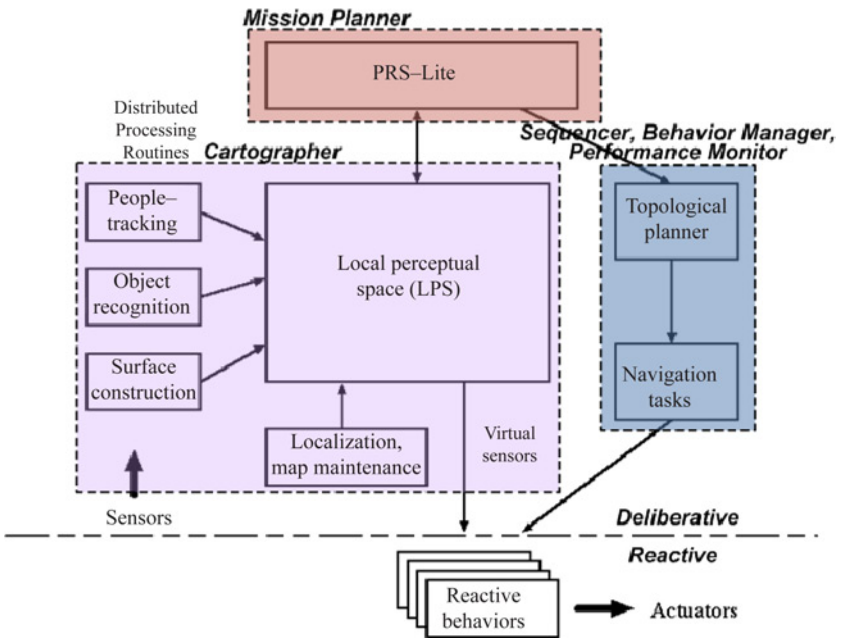

Fig. 5 The Saphira architecture (bi-level).

needed for certain applications, such as the DARPA Urban Challenge. Obstacle avoidance and lane maintenance on a 1.8 ton commercial vehicle moving at 30 mph requires performance guarantees that are often difficult to achieve with behavior-based motion control.

To address this problem, VictorTango has incorporated technological advances in model-predictive motion planning into a *tri-level* Hybrid architecture. In this tri-level approach, there still exists a deliberative layer at the highest level, responsible for such tasks as mission planning and map making. These modules then provide intermediary goals to a reactive, behavioral component that is then sandwiched by yet another low-level deliberative layer. The end result is a *deliberative–reactive–deliberative* progression. The low-level deliberative layer is strictly responsible for motion planning and sends the final output to the vehicle 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, the low-level path planner continually replans at a high rate.

VictorTango's tri-level software architecture is shown in Fig. 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. Similar to Saphira, all sensor data are 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 the global world model and are provided to the behaviors and planning components as *virtual sensors*. Perception can therefore be seen as a parallel process, and the same representation of the world is used by all three decision-making layers.

Because a deliberative approach is being used for low-level obstacle avoidance, dynamic stability, and lane maintenance, the scope and responsibility for the behavior-based software agent shifts upward. The need now exists for a behavioral component capable of complex decision making to complete a variety of abstract, multifaceted, temporal problems. Specifically, this behavioral module is responsible for providing contextual intelligence, monitoring the situation at hand, and taking into account the changing goals and subgoals of the vehicle. Through a method of

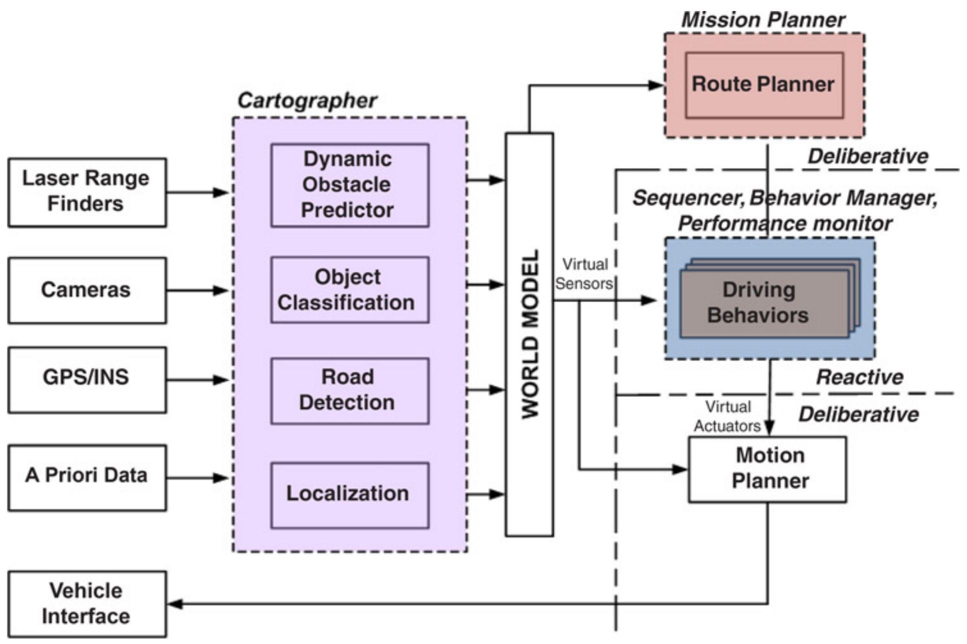

Fig. 6 The VictorTango architecture (tri-level).

behavior coordination, acceptable emergent behavior given the current situation must be produced through the issuing of high-level, hardware-independent motion commands, or *virtual actuators*.

B. Perception

Odin's perception is handled by a collection of software modules intended to interpret and translate specific sensor information into a small set of virtual sensors, or sensor-independent perception messages, for the planning modules. These virtual sensors describing the location of lanes, drivable area, static obstacles, and dynamic obstacles, and the vehicle's pose compose the world model that the planning software acts upon. By defining virtual sensors that are not dependent on specific hardware, the addition of another sensor or sensor modality affects only the perception software, improving the overall performance of Odin while enhancing the reusability of the decision-making software for future projects.

1. Decoupled Local and Global Frames

Odin's localization software reports two decoupled position solutions: a global positioning system (GPS)-based global position and an inertial local position. Used in primarily referencing the RNDF, the global position relied primarily on a COTS-filtered GPS/INS (inertial navigation system) solution. This solution was fused with measured wheel speeds and steering angle using an extended Kalman filter designed for Odin, based on standard practices [13]. Implemented to assist the GPS/INS solution during times of poor GPS signal or GPS outages, the filter also adds to the scalability of the module because the inherent nature of the filter allows for future measurements (expansion of perception capabilities) and redundant information (more robust through statistical weighting).

The second position solution is an entirely inertially based local position. Starting from an arbitrary origin, Odin tracks its location through the local frame using wheel speeds, steering angle, and inertial measurements. The error associated with this frame is typically within 2.6% of the distance traveled, an error accumulation not significant within the sensor range. The goal of this frame is to provide a smooth and continuous position estimate for obstacle tracking and short-range planning that is devoid of the potential GPS jumps or errors. Such discontinuities, known as "GPS Pop", can be disastrous to object tracking and prediction algorithms that compare sequential frames of data. An instantaneous local-to-global transformation calculated with the local solution allows locally detected objects or roadway information to be transformed into the global frame for comparison against the RNDF. For more information on the localization software, see [14].

2. Road Detection

The Road Detection software component determines the roads and zones that are available for navigation, providing its output in two messages: lane position and drivable area coverage. For each lane, the lane position message provides the position and width of the travel lane, as well as all possible branches (smooth paths to other lanes) at upcoming intersections. Drivable area coverage expresses road and zone data as a vehicle-centered, 2D binary map; for example, in a zone, a parking lot island would appear as undrivable area. Both outputs are expressed in the local frame. Road data are used in Object Classification for filtering out objects not close to a road, in the Dynamic Object Predictor for calculating likely trajectories of other traffic, and in Driving Behaviors and Motion Planning for planning. For more information on the Road Detection software, see [1].

3. Object Classification and Dynamic Object Predictor

Object Classification is responsible for merging data from multiple sensors to detect and classify nearby objects. Objects are classified into two categories: static obstacles that do not move, and dynamic objects that are in motion or have the potential for motion; for the Urban Challenge, the only dynamic objects Odin was expected to encounter were other vehicles. Objects are provided to downstream software modules as a list of static objects (with profile points), and a list of dynamic objects (with a length, width, heading, speed, stopped time, and profile points). All locations are transformed from vehicle to local frame before transmission.

While static objects are passed directly to Motion Planning, the list of dynamic objects (vehicles) is passed to a Dynamic Object Predictor, which is responsible for classifying each vehicle in a lane and determining likely paths for each vehicle, incorporating lane position information from Road Detection. A vehicle's path is determined by examining its motion history and relevant lane position data for the segment it is on; if there are no available lane data, as is the case inside a zone, the path is projected based on the vehicle's current motion. Once the set of paths is calculated, it is appended to the vehicle's data as a series of discrete positions, each with an approximate arrival time. The final dynamic object list is used by Driving Behaviors and Motion Planning for traffic behavior and obstacle avoidance. For more information on the Object Classification software, see [15].

4. Perception Focus

In certain situations, such as merging or exiting across an oncoming traffic lane, it is beneficial to have perception extend its sensory range in one or more directions, at the expense of another direction. To enable this, the road detection, object classification, and dynamic object predictor modules can receive a "set perception focus" message instructing them to extend focus left, right, rear, or forward. This feature allows the autonomous vehicle to effectively "look both ways" while merging. Certainly, one solution could have been to extend the perception range in all directions at all times, however to reduce unnecessary processing and due to a 64-object limit of the IBEO laser scanners, it became important only to extend the range in certain directions depending on the situation, temporarily sacrificing range in another direction.

C. Decision making

VictorTango's decision-making software is broken into four main parts: a deliberative Route Planner that establishes high-level mission goals; a reactive Driving Behaviors component to follow traffic laws by spooling short term goals with constraints; a deliberative Motion Planner that establishes a safe path through the local area to achieve the short-term goals; and a Vehicle Interface that commands the actuators to execute the plan.

1. Deliberative Route Planner

The Route Planner software module is responsible for choosing which roads are traveled to complete a mission. The only software component on Odin that runs on demand rather than periodically, the Route Planner generates a route whenever a new MDF is loaded or if the current road segment is completely blocked, requiring a new route. The output of the Route Planner is a list of waypoints that are exits, entrances or checkpoints. Any waypoints along the road lane are omitted, as the Route Planner does not choose the lane of travel if there are multiple lanes in a road.

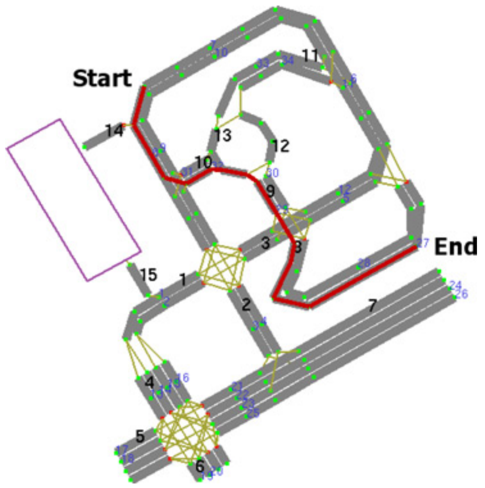

Fig. 7 The Route Planner selects the road exits and entrances used to complete a mission, shown as highlighted route.

Checkpoints, however, must still be specified to ensure the vehicle is in the proper lane when passing a checkpoint. An example Route Planner output is shown in Fig. 7.

Only a limited subset of world information is considered by the Route Planner. The Route Planner uses the road information contained in the RNDF file, speed limits defined in the MDF file, and road blockage reports from the Driving Behaviors software module rather than direct obstacle information.

The implementation of the Route Planner is based on an A* graph search. The exit and entrance pairs of the road network are used to form a graph and an A* search is used to minimize travel time from the current vehicle location to the mission checkpoints. Travel time is estimated by applying the MDF speed limits to the distances between lane waypoints. To account for time at intersections and reduce traffic interactions, time penalties can also be applied to certain events. For the Urban Challenge, the time penalties used were: 30 s for any exit, 180 s for a U-turn, and 400 s for entering a zone. Another final event parameter was the distance needed for a lane change, which was set to 20 m. For example, if a road had two forward travel lanes, and Odin entered the right lane, the Route Planner would only connect exits in the left lane that were at least 20 m down the road. The completed route is then output to the Driving Behaviors module and the Route Planner remains idle until a new MDF is loaded or a blockage is encountered.

2. *Reactive Driving Behaviors*

Driving Behaviors is responsible for maintaining situational awareness, obeying the rules of the road, and interacting with other traffic while achieving the mission. This includes choosing which lane to drive in, deciding when lane changes are necessary, determining and respecting the progression order at intersections, handling blocked roads, and reacting to other unexpected traffic situations. To accomplish this high-level task, Driving Behaviors spools short-term goals and motion guidelines to the Motion Planner via a set of virtual actuators that make up a behavior profile.

Messaging: At its core, the Behavior Profile sent to Motion Planning comprises six target points, expressed in local coordinates. Each point contains the lane and lane branch (for intersections), and an optional Stop flag and heading criteria. A desired maximum speed, travel lane, and direction (Forward, Reverse, Don't Care), are also included. Finally, the Zone and Safety Area (intersection) flags can be turned on to enable different modes within Motion Planning. As the vehicle progresses, Driving Behaviors receives feedback from Motion Planning on the achievability of the currently commanded goals, as well as notification when a target point is achieved. Driving Behaviors also commands the turn signals and horn directly to the Vehicle Interface and specifies a perception focus to the perception modules.

Implementation: As mentioned in the architecture overview, the role of the reactive, behavior-based layer has shifted upwards to bridging the gap between high-level mission planning and low-level navigation. Specifically, the Driving Behaviors module provides two important aspects of embodied artificial intelligence (A.I.), contextual intelligence and emergent behavior. 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 Route Planner, as well as the current environment, as defined by the virtual sensors. Such insight is important for performance monitoring and self awareness along with the ability to balance multiple goals and subgoals. 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 multi-agent systems as well.

In total, Driving Behaviors is tasked with selecting the most ‘appropriate’ actions, taking into account the mission plan, sensed traffic, sensed road blockages, and the rules of the road. The process of deducing the most appropriate action is known as the Action Selection Problem and solutions to this problem are known as Action Selection Mechanisms (ASM). Existing ASMs, according to [16], fit in either the *arbitration* or *command fusion* class, as shown in Fig. 8.

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” [16]. 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. Command fusion ASMs, on the other hand, “allow multiple behaviors to contribute to the final control of the robot” [16]. Rather than being concerned with selecting appropriate behaviors, command fusion ASMs let all behaviors run concurrently, and then rely on a fusion scheme to filter out insignificant behavioral outputs. Command fusion ASMs are therefore typically described as being flat. As multiple behaviors can end up desiring the same control, these ASMs present novel methods of collaboration amongst behaviors. This cooperative approach, rather than competitive, can be extremely useful in situations with multiple, concurrent objectives. On the other hand, 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 task.

The novel, behavior-based architecture developed by VictorTango for the Driving Behaviors component preserves the strengths of both Arbitration and Command Fusion action selection mechanisms (ASMs) [17]. 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.

Arbitration Mechanism: Specifically, a state-based, hierarchical, arbitration ASM is used for behavior selection. This method was a hierarchical network of finite state automata (FSA), which can be referred to as a hierarchical state machine (HSM). 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 [18]. Within the behavioral HSM, each individual state machine refers to a single behavior. Behaviors are organized in the hierarchy with more abstract behaviors higher in the tree, and more physical behaviors lower in the tree. Each behavior, or node, in the tree is responsible for determining which of their subbehaviors should be activated. This is determined by each behavior’s internal state and is not limited to only one subbehavior. All behaviors have implied relationships based off of their position within the hierarchy tree. Behaviors can have parent–child relationships, or sibling relations, but is important

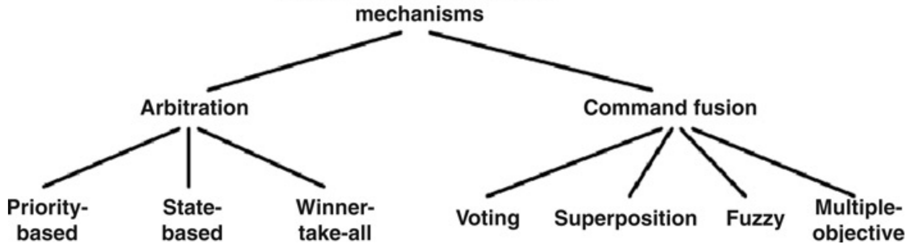

Fig. 8 Pirjanian’s taxonomy of ASMs.

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 [5], 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. The behavioral HSM used on Odin for the Urban Challenge can be seen in Fig. 9.

All behaviors are classified as a command and/or decision behavior. A command behavior produces 1 or more virtual actuator (VA) commands, and a decision behavior results in the activation of a lower subbehavior (i.e. not a leaf node). Each behavior can 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 ((VS) 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 subbehaviors $sb_i \in SB$ to activate when in that state. Transitions between control states occur 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 subbehaviors should be running. Should a subbehavior complete its state sequence and have nothing to do, it will remain in an idle state and not compete for control of any VA.

Command behaviors in Fig. 9 are labeled as "drivers". Level 1 behaviors are primarily concerned with traversing the course solely on a priori information. These behaviors use the given RNDF and other a priori information to execute as if no surprises in the world exist. Level 2 behaviors are then responsible for handling events that cannot be planned for, such as intersection traffic, disabled vehicles, and roadblocks.

Command Fusion Mechanism: With all the command behaviors defined as individuals, it is necessary to look at how these behaviors interact as a group. Because 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_{tane} of the Passing Driver disagrees with π_{tane} of the Route Driver, some form of resolution is needed. When this situation occurs, it is the responsibility of the command fusion mechanism to produce the final desired lane that will be bundled in the behavior profile.

Once the final set of behaviors has 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 use many existing techniques [5,6,19] or implement a novel method. What dictates the choice should be the robot

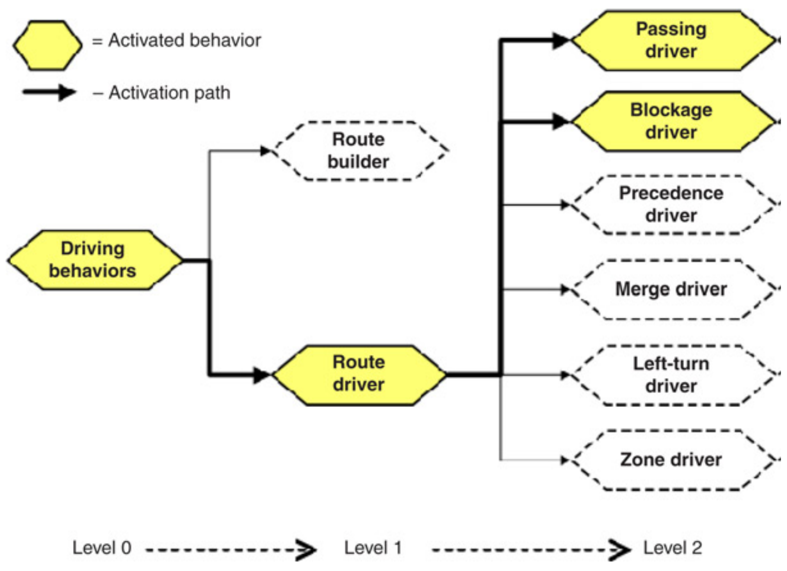

Fig. 9 Behavioral HSM used driving in an urban environment.

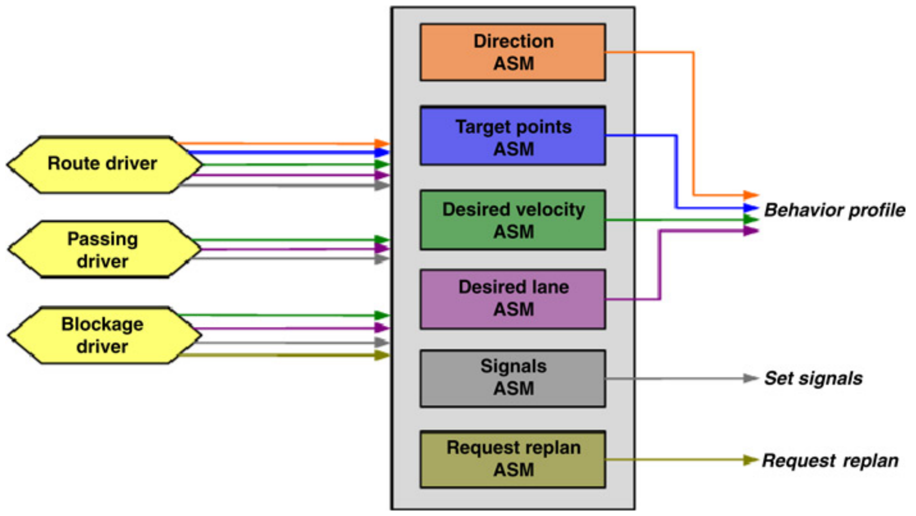

Fig. 10 Command Fusion ASMs used on Odin.

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 10 illustrates the command fusion structure used in the DARPA Urban Challenge. For simplicity, only the Route Driver, Passing Driver, and Blockage Driver are activated in this example. 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.

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. Because of the rule heavy nature of urban driving, and the presence of well-defined individual maneuvers such as passing, a nonsuperposition-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 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.

Hierarchy with Parallelism: VictorTango's approach to behavioral programming presents the idea of using hierarchy *with* parallelism, when typically these features are mutually exclusive. It has been shown in [20] that the major bottleneck in developing behavioral intelligence is not selecting the best approach or architecture, but developing the *correct version* of this approach. While complexity is needed for multifaceted problems, reducing complexity is important for making the robot designer's job simpler. Furthermore, it is not enough that a behavioral system be able to do a lot of things, it is equally important that they do all those things right, and at the right times.

In real-world applications with major repercussions for incorrect behavior, performance predictability can be paramount. The ability to hand code behaviors and ignore certain perceptual triggers at certain times is extremely useful. Hierarchy takes advantage of *selective attention* to make this hand-coding of behaviors possible and practical. Yet at the same time, complex combinations of behaviors are important for developing higher level intelligence. Combining hierarchy with parallelism in the method presented here provides important flexibility to the behavioral programmer. Situations in need of predictability can be catered to, while other situations can still take advantage of complex, parallel, combination schemes. This approach balances quantity and complexity with design practicality.

3. *Deliberative Motion Planning*

The Motion Planning software module is the final layer of planning before low level vehicle control. Motion Planning controls the speed and path of Odin to stay in the specified lane, avoid obstacles, and obey limits set by higher level software components. Motion Planning receives all perception messages, but operates entirely in the local coordinate frame and never needs to globally reference any positions. Motion Planning receives commands from Driving Behaviors in the form of a behavior profile message. The behavior profile sets maximum speeds, specifies the lane of travel, and can also specify a desired heading for situations like parking where Odin should be aligned with the parking spot. Because Motion Planning receives data from many different sources at different rates, the Motion Planning software runs with a faster cycle time than most other components, typically 80 ms. This faster update rate allows Motion Planning to incorporate the latest data as it continuously replans.

Messaging: Motion Planning produces motion commands called motion profiles that are sent to the Vehicle Interface and also can send messages to Driving Behaviors if the desired lane is not safely achievable. The motion profile provides a platform independent message that specifies the path of travel, leaving the Vehicle Interface to translate into vehicle actuation. In practice both the Motion Planner and the Vehicle Interface will have knowledge of the vehicle platform dynamics, but a platform independent message allows reuse of the message in other systems, with the Vehicle Interface incorporating additional parameters such as understeer that do not need to be planned for in the Motion Planning level. The motion profile message sets both the path and the speed, as shown in Table 1.

The motion profile message contains a series of commands where each command has a time duration. The Vehicle Interface executes the first command until the time duration expires, and then begins to execute the next command. This process repeats until a new motion profile message is received, or until there are no more commands to process. When a new motion profile message is received, the previous message is completely discarded and the new profile is executed immediately. If the time duration of the last command in a motion profile expires, the Vehicle Interface enters a safety mode and brings Odin to a stop. While executing a command, the Vehicle Interface attempts attain the desired velocity as aggressively as possible without violating the acceleration limit. The velocity is held at the desired velocity until the time duration expires, and the vehicle may not even reach the desired velocity if the time duration or acceleration is set too small. The curvature works in a similar way, where the curvature is held at the desired set point until the time duration expires, limited by the specified rate of change.

Implementation: The implementation of Motion Planning divides the module into two distinct software components: the Speed Limiter and the Trajectory Search. The software is split to separate the problem of how fast to go from where to go. An example of this is when a slower moving vehicle is in front of Odin, in which case Odin should continue to follow the same path as if the road were clear, but must slow down to react to traffic. The Speed Limiter monitors the road for traffic and stop lines and sends the Trajectory Search component a maximum speed and the ID of any vehicle that is affecting the maximum speed. The Trajectory Search will ignore the traffic specified by the Speed Limiter and plan a safe path using the lane data, static obstacles and other dynamic obstacles. Trajectory Search still plans both a path and final speed, as slower travel may be necessary for sharper turns. Figure 11 shows the inputs and interactions of Motion Planning.

The Speed Limiter starts by identifying which dynamic obstacles could enter Odin's path. The Dynamic Obstacle Predictor assigns lanes to each obstacle, which the Speed Limiter compares to the future path of Odin. To ensure safety, any part of a vehicle within a lane is sufficient to be assigned to the lane. The Speed Limiter calculates the minimum clearance to pass another vehicle within the same lane, allowing Odin to pass parked cars on the side of

Table 1 Contents of motion profile message

Field #	Name
1	Number of commands, X
$1 + 5x + 1$	Desired velocity (m/s)
$1 + 5x + 2$	Acceleration limit (m/s^2)
$1 + 5x + 3$	Path curvature (1/m)
$1 + 5x + 4$	Rate of curvature change (1/m/s)
$1 + 5x + 5$	Time duration (s)

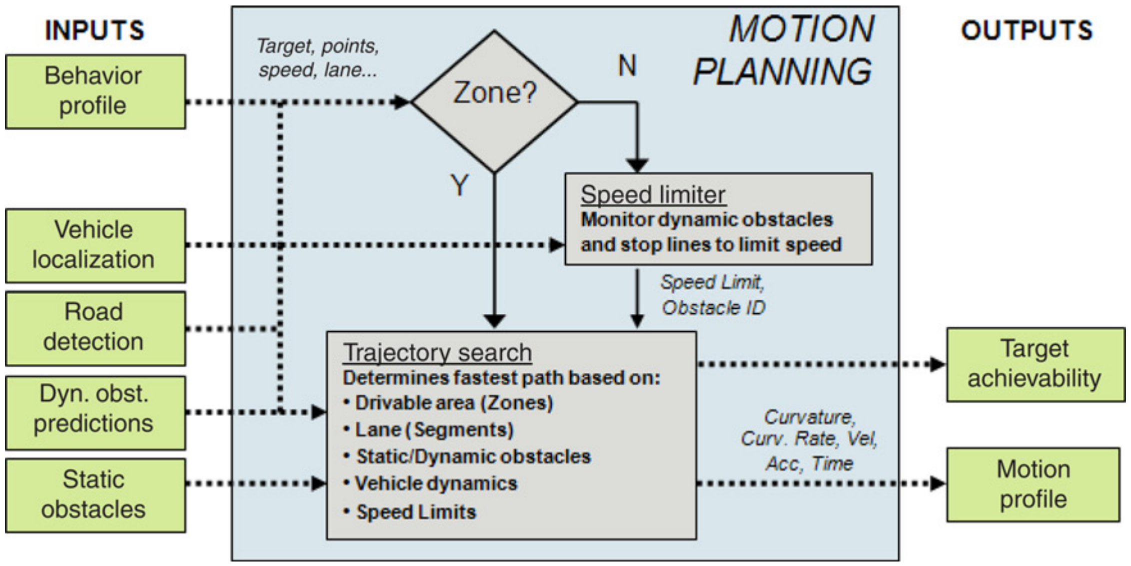

Fig. 11 Software flow diagram of Motion Planning component.

a lane, slowing down if there is a tight clearance, but not attempt to pass a slower vehicle that is mostly within a single lane.

The Trajectory Search component plans the future path of Odin by creating a cost map using road and obstacle data, and searching precomputed trajectories to form a continuous path. Trajectory Search uses a database of simulation results that form a coverage map of future vehicle occupancy for a given initial state and command. The search process consists of selecting trajectories using an A* search and evaluating for any collisions with obstacles or lane boundaries until the goal criteria is met, such as a distance down the lane. The search produces a smooth path by trying to minimize the travel time necessary. Obstacles are entered into the cost map with a high cost at the obstacle location and radiate decreasing cost. This favors keeping a minimum distance from all obstacles, but allows tight maneuvering if all other options have been exhausted. For more information on this implementation, see [1].

4. *Vehicle Interface*

The VictorTango architecture is designed such that the vehicle is a separate unit from the perception and planning modules. The design of the architecture provides several advantages by separating the hardware interface and low-level control systems from the planning modules. The independence of the low-level control systems removes the need for the motion planning algorithm to have a detailed internal model of the vehicle dynamics. A basic linear model of a generalized Ackerman-steered vehicle can be defined using only a few basic performance parameters for use in Motion Planning. Generalized motion parameters such as path curvature, speeds, and rates of change can be commanded to the vehicle interface. The vehicle interface can then be tuned to convert these generalized parameters to actuator commands that correct for the true non-linear dynamics of the vehicle.

The use of an independent vehicle interface commanded by generalized motion parameters also enhances the future reusability of the planning algorithms. The planning modules could be transferred to another similar autonomous platform by changing only a few basic parameters describing maximum vehicle steering and speed capabilities. This has been successfully demonstrated with Odin's software, which has been reused on other vehicles for other projects after the Urban Challenge.

D. Health Monitor

For Odin to operate completely autonomously for hours on end, with absolutely no human interaction, it becomes critical to implement a strategy for fault detection and health monitoring. Even seemingly simple errors such as a brief communications timeout can result in a catastrophic vehicle crash if the proper precautions are not taken.

In Odin's architecture, health monitoring is handled at two levels. First, all software was designed with safe failure modes in mind; for example, if the Motion Planner is receiving no speed feedback, it defaults to commanding a zero speed. Second, each software module is responsible for detecting errors relevant to itself and reporting them to a central Health Monitor. These errors may range from an RNDF parsing error, to a cycle time error, to an unsatisfactory update rate in a message being received from another module, and are reported with an error code, a human-readable string, and a criticality (warning, serious error, and catastrophe). Based on the cumulative set of errors, the Health Monitor decides whether the vehicle can proceed given its current state, and, if necessary, issues a software pause command to the Vehicle Interface (zeroing the speed while retaining steering control, emergency flashers on). Once the fault has been cleared, Odin resumes normal operation.

In certain driving situations however, such as crossing a moving traffic lane when entering a main road, it is more dangerous to stop immediately in the traffic lane due to an error. In this case, Driving Behaviors can issue a temporary urgency override, disabling certain error-related software pauses at the Health Monitor. This urgency override remains engaged until the vehicle is safely within its own travel lane, or a timeout expires.

One key example of health monitoring manifested itself once during the Urban Challenge. During testing it was observed that the IBEO sensors would occasionally time out, and to fix the problem that sensor's Electronic Control Unit (ECU) had to be restarted. While this 90-s operation is taking place, Odin's ability to perceive objects is severely limited, making it unsafe to continue. However, the Object Classification module does not have the ability to control the vehicle's motion, and the Vehicle Interface has no direct way of identifying problems with sensors or software. To address this problem, the Object Classification module reported an error to the Health Monitor, which commanded the Vehicle Interface to pause Odin until the sensor had been restarted and the error cleared.

E. Communications

The primary communications backbone on Odin is a gigabit Ethernet network for interprocess communication between software modules, with an additional Controller Area Network (CAN) network for low-level communications confined to the Vehicle Interface. In addition, most sensors were also attached to the Ethernet network, either natively (the IBEO ECUs), or via MOXA Serial-to-Ethernet converters.

1. JAUS Implementation

For interprocess communication between software modules, Odin employs SAE AS-4 JAUS. Each of the major software modules (Road Detection, Motion Planning, Health Monitor, etc) is implemented as a standalone JAUS component/service that interacts with other software by passing JAUS messages via the local JAUS Node Manager for that particular computing node (Linux virtual machine, Windows server, CompactRIO, TouchPanel, or developer's laptop), which routes all message traffic within the computing node or to other Node Managers within the vehicle subsystem.

Components send and receive the JAUS core messages to discover other modules, determine their functionality, and request information from them via periodic or change-based events, enabling dynamic configuration. This allows a software module to be run on a developer's laptop connected to the vehicle network or to be seamlessly shifted to another computing node. In addition, a new module requiring information from an existing module could simply establish a periodic event, without requiring recompile or reconfiguration of existing software. Figure 12 shows the core message sequence that allows for dynamic configuration to take place. In this example, Driving Behaviors is able automatically to discover Localization and create an event to receive Local Position at a rate of 10 Hz.

Where possible, the Platform messages in the JAUS standard, such as Report Velocity State, were used to allow for future interoperability with other JAUS-enabled percepts, planners, platforms, payloads, or control stations. Due to the cutting-edge nature of the urban driving problem however, JAUS does not yet provide standard messages for everything, and it was necessary to implement many experimental messages. Approximately 40 experimental messages were developed to pass data such as lane position, static obstacles, dynamic obstacles and predictions, health monitoring errors, turn signal commands, and other data related to urban driving. With active developers in the SAE AS-4 standards development committee, VictorTango is working to integrate a subset of these messages and protocols into a future version of JAUS.

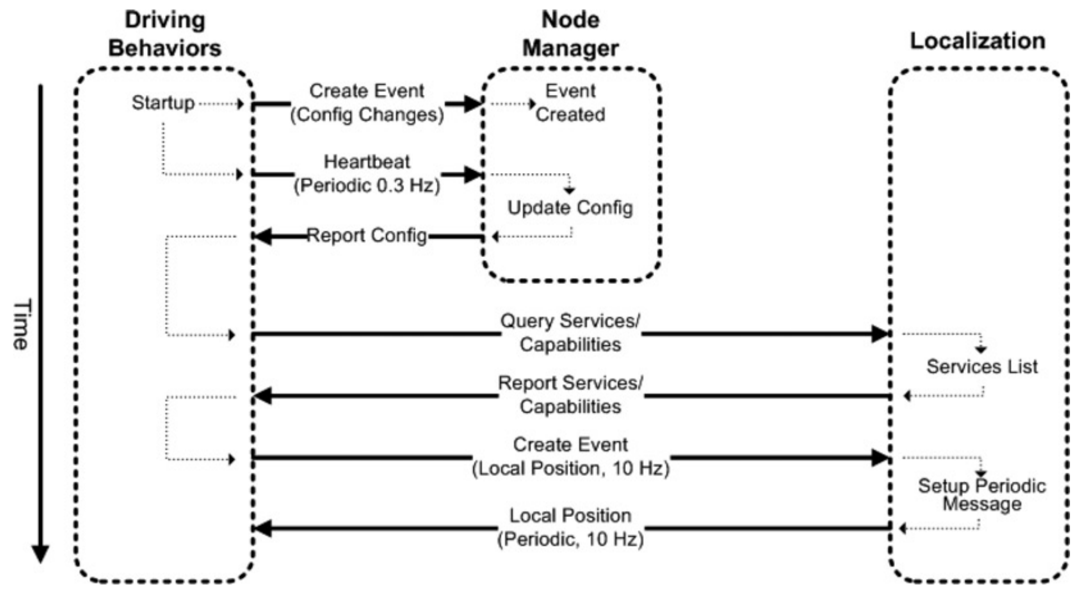
Driving Behaviors
Node Manager
Localization
Time
Startup
Create Event (Config Changes)
Event Created
Heartbeat (Periodic 0.3 Hz)
Update Config
Report Config
Query Services/ Capabilities
Services List
Report Services/ Capabilities
Create Event (Local Position, 10 Hz)
Setup Periodic Message
Local Position (Periodic, 10 Hz)

Fig. 12 Detailed message flow during dynamic configuration.

The JAUS implementation on Odin made use of a well-tested JAUS Toolkit previously developed by TORC. In doing so, the communications infrastructure provided key development features such as a generic integrated data-logging and replay system. Additionally, the use of JAUS facilitated scalable simulation tests, from the full software suite to a condensed version running on a single laptop computer. But most important, the use of JAUS coupled with careful architecture design allows for the rapid, immediate reuse of Odin's software on other autonomous vehicle projects, a step critical to the acceleration of unmanned systems development.

2. *Integrated Simulation*

Simulation played a crucial role in the software development process of Odin, especially in the development and testing of the planning software. As such, the communications implementation for Odin was designed to facilitate simulation. A custom simulation tool, shown in Fig. 13, was developed using the same messaging architecture as Odin. This tool, combined with the automatic configuration afforded by the JAUS implementation, allowed software components to be transitioned from simulation testing to deployment on Odin without any configuration changes.

During full-scale simulation testing, all sensing and perception software messages were generated by the simulator, and planning software would be run as normal. To isolate planning problems from perception bugs, the simulator produces perfect sensing data using the architecture defined sensor-independent perception messages to report data such as position, static obstacles, and dynamic obstacles. Successful decisions made by the planning software in the simulation environment would not guarantee success in the real world, however any failures experienced in simulation would most certainly show up during more time-intensive real-world testing.

The simulator was designed to be modular, allowing different vehicle types to be equipped with custom defined sensors. This modularity allowed for the creation of a vehicle type that directly receives position and orientation updates from the localization software on Odin rather than simulating the motion of Odin. This would allow the use of the simulator as a visualization tool during tests. The same updated vehicle model could also be equipped with virtual sensors just as the fully simulated vehicle model. This allowed hardware in the loop simulation tests, where Odin would drive in an empty parking lot with no obstacle detection software running. Instead, the simulator would supply static and dynamic obstacles, providing a safe way to test dangerous situations such as traffic not obeying intersection precedence, or minimum stopping distance. Early in software development, any new situation involving traffic would be tested in this manner before attempting with human traffic drivers.

III. Results and Conclusions

The VictorTango software architecture was thoroughly tested and implemented on Odin, one of 35 vehicles invited to participate in the DARPA Urban Challenge (DUC) National Qualifying Event. From these 35, a total of 11 teams were selected to participate in the Urban Challenge Event. Only six teams managed to complete the course, with the top three places going to Tartan Racing, Stanford Racing Team, and team VictorTango. During the competition, Odin ran for several hours without any human intervention, negotiating stop sign intersections, merging into and across traffic, parking, and driving down multilane roads in the presence of forward and oncoming vehicles. The VictorTango architecture handled all these technical challenges well, proving its overall merit as a field-capable architecture.

Beyond the DUC performance metric, this section will examine more closely the performance of the VictorTango architecture. Several common situations will be examined to demonstrate the flow of information and sequence of decision making across software modules. An analysis of timing and message latency and their effect on vehicle reaction times will be given, followed by a discussion of future work and areas for improvement.

A. Situational Examples

The interactions of the multitude of different software modules on Odin are illustrated here in four simple examples: normal driving on an empty road, handling an intersection, encountering a blocked road, and navigating a zone. Each example showcases a different type of interaction.

1. Driving a Road

Before driving, an RNDF and MDF are loaded via a touch-screen user interface located in the vehicle. Upon receipt of the MDF, Driving Behaviors locates itself in the RNDF, chooses a waypoint to begin the route from, and requests a route (in the form of exits and entrances) from the Route Planner. Driving Behaviors then fills in intermediate waypoints between the exits and entrances specified by the Route Planner, pre-assigning a driving situation to each point. Driving Behaviors then spools a small set of waypoints from this predetermined route to Motion Planning, along with a set of constraints such as desired lane, desired speed, direction, etc. As Motion Planning achieves the commanded points, it reports them achieved and Driving Behaviors shifts the spool forward, activating more specific specialized behaviors depending on the driving situation, to handle traffic or blockages requiring departure from the predetermined lane. These drivers simply override the default behavior by modifying the lane, speed, direction, or target points, and may on occasion request a new route from the Route Planner.

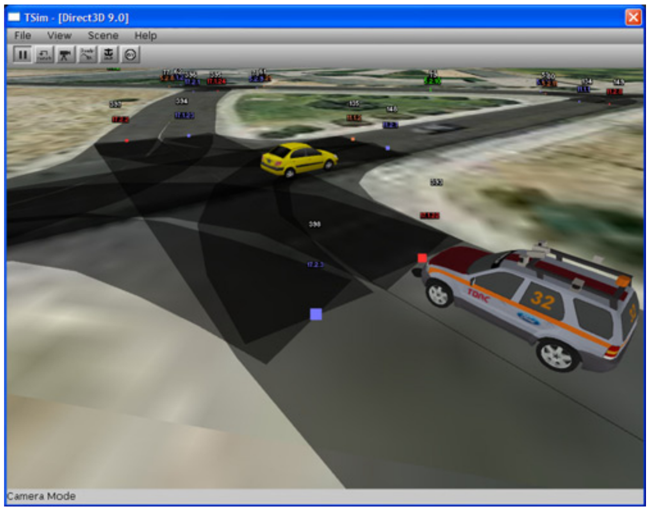

Fig. 13 TSim simulator showing Odin at an intersection.

2. Intersection

Nearly every software module plays an important role at intersections. On the perception side, Road Detection describes the form of nearby lanes and their branches through the intersection, Object Classification identifies and classifies dynamic and static objects, and the Dynamic Object Predictor predicts likely paths for nearby dynamic objects. For planning, Driving Behaviors examines the dynamic objects and predictions to properly observe stop-sign precedence and merge into or cross moving traffic lanes, respecting traffic laws while handling abnormal traffic behavior as necessary. Finally, Motion Planning is responsible for achieving the directives from Driving Behaviors and reporting when its commanded goals are not achievable, while providing basic reflexive avoidance of dynamic and static obstacles.

In the example of approaching a stop sign at an intersection with both thru traffic and stop sign precedence (see Fig. 14), the following sequence of interactions take place. As Odin approaches the intersection, a Precedence Driver and Merge Driver are activated within Driving Behaviors; if Odin is making a turn, Driving Behaviors instructs the Vehicle Interface to activate the turn signals. When Motion Planning achieves the stop point and reports it achieved, Driving Behaviors begins waiting for any other vehicles at other stop signs. Once it is Odin's turn to proceed, Driving Behaviors instructs the perception modules (Road Detection and Object Classification) to extend their range along the roadway, depending on the direction of approaching traffic. After a safe opening into traffic has been found, Motion Planning is given a nonzero speed, and begins to traverse the intersection. To prevent the Health Monitor from pausing the vehicle due to an error while crossing a traffic lane, a temporary urgency override is issued to the Health Monitor once the vehicle is moving fast enough that it cannot safely stop without protruding into the lane. At this point, the vehicle has committed to completing the merge, and the perception modules return to normal sensing range. Finally, once Motion Planning reports the next target point achieved, the vehicle returns to its normal road driving situation. This messaging sequence can be seen in Fig. 15.

3. Blocked Road

In this example, Odin encounters a blockage spanning all available travel lanes of the current RNDF segment. As only Motion Planning knows the exact kinematic constraints of the vehicle, it is responsible for determining that Odin cannot proceed in the lane being commanded by Driving Behaviors, stopping the vehicle, and reporting the target points as unachievable. A Blockage Driver within Driving Behaviors monitors the reported achievability from Motion Planning; once the lane is unachievable for sufficient time, Driving Behaviors will try to command a lane change. If all available lanes are reported unachievable and a U-turn is legal, Driving Behaviors alerts the Route Planner of the blockage and requests a new route. After adjusting the graph, the Route Planner determines a new route to achieve the remaining checkpoints in the MDF. Once a new route, complete with a U-turn, has been assembled in Driving Behaviors, the direction is constrained to "Don't-Care", and Motion Planning is entirely responsible for getting Odin into the new lane, but is free to examine paths in reverse. After Motion Planning completes the U-turn, it reports the goal achieved and Driving Behaviors returns to the default driving situation, with direction "Must Be Forward". This messaging sequence can be seen in Fig. 16.

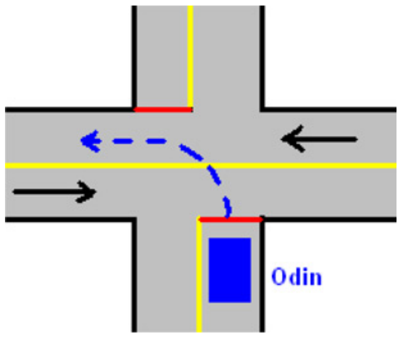

Fig. 14 Intersection with precedence and merging.

4. Parking Lot

The zone navigation problem presents a very different challenge from navigating lanes; however Odin handles it in a similar fashion. One primary difference is that Motion Planning alone is responsible for all dynamic obstacle avoidance in zones, a decision made due to the largely unknown environment in a zone, and the desire to keep Driving Behaviors independent of the exact mobility constraints of the vehicle. Motion Planning also makes more extensive use of the drivable area map from Road Detection, for avoidance of islands.

To navigate through the zone, to parking spots or the zone exit, target points with constraints are spooled from Driving Behaviors based on a graph of the accepted travel patterns within a parking lot (around parking rows and islands). Motion Planning is then responsible for following these goal points in similar fashion to a roadway lane,

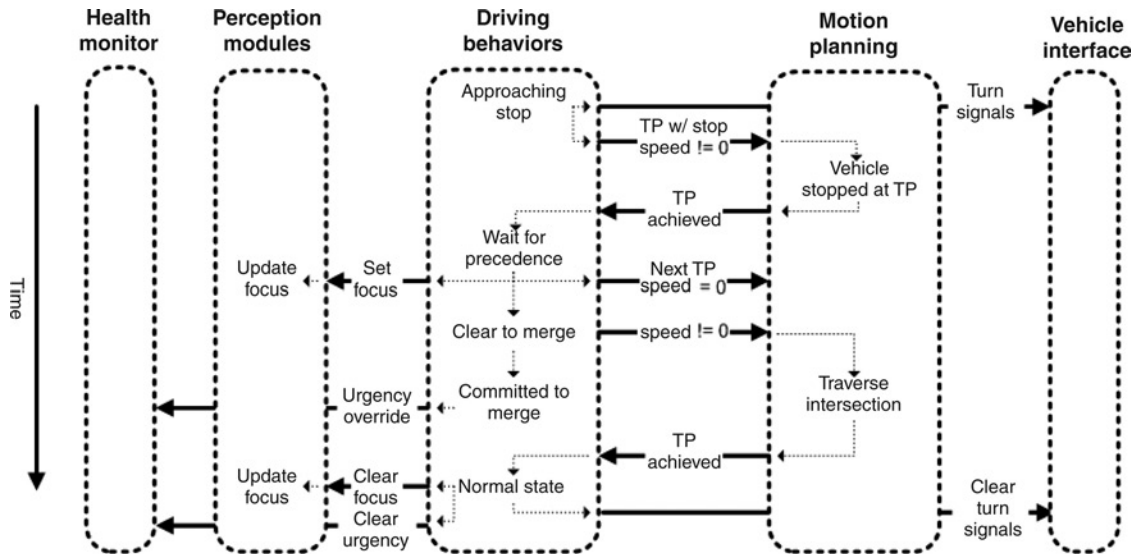

Fig. 15 Messaging sequence when merging at an intersection.

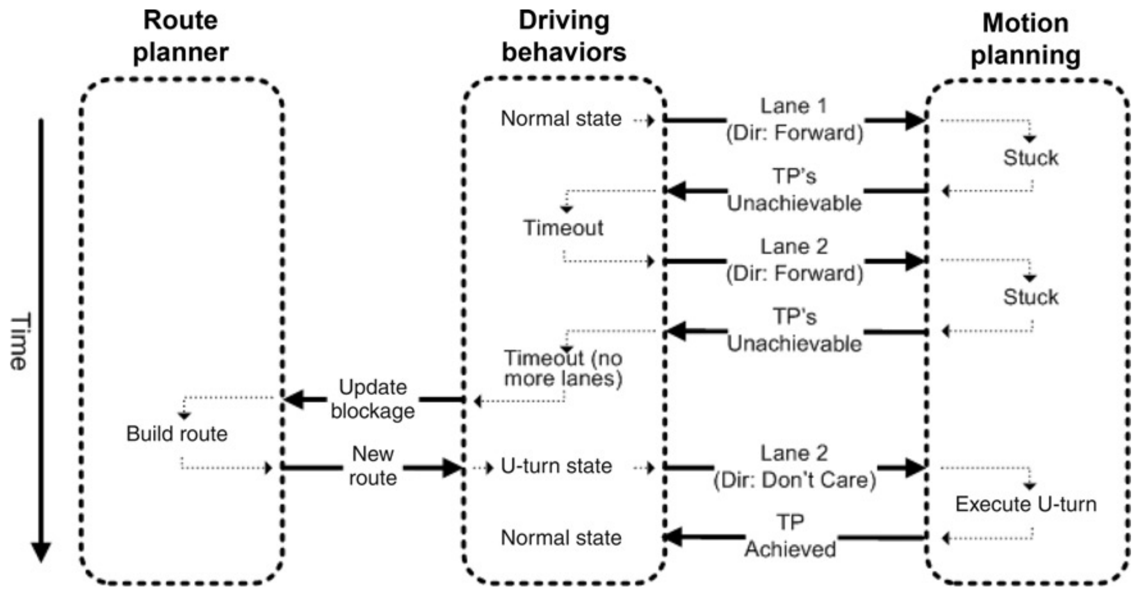

Fig. 16 Messaging sequence when encountering a roadblock.

with the additional freedom to move in reverse. A robust set of gracefully degrading, increasingly generous recovery modes ensures that Motion Planning will almost always never stop permanently, however without knowledge of the RNDF except as described by Road Detection, it is unaware of alternate paths to the exit or parking spot. To address this, a progress monitor within Driving Behaviors tracks the vehicle's progress and reported achievability from Motion Planning, and if necessary, can disconnect a segment of the graph and re-plan through the zone.

B. Analysis of Latency and Timings

When designing a software architecture for a task as complex as the Urban Challenge, care must be taken to ensure that critical information such as the location of obstacles reaches the planning software before it is too late to react. This is especially important for Odin as all sensor data are processed into sensor independent messages, adding additional layers of processing and latency. Some effects of latency can be mitigated by the choice of the message data used. For example, static obstacles are transformed from vehicle frame to a locally referenced frame before transmission. Once in a local reference frame, any delay in transmission to Motion Planning will not affect accuracy, it will only reduce the maximum distance at which the software can react to obstacles.

The JAUS protocol allows the specification of messaging rates for periodic data. To reduce development time and network bandwidth, the rate of transmission for each message was set to 10 Hz initially, and increased as necessary. Table 2 presents the processing cycle time of major software components and the messaging rate used per software output. This table is not exhaustive and omits messages such as turn signal control or health monitoring messages. Messages that have a messaging time of 0 ms are ones that are sent immediately, rather than read and forwarded at a periodic rate.

Table 2 Software latency and timing by message

Software module	Processing time (ms)	Data	Destination	Messaging time (ms)
Vehicle Interface	10	Wheel speed	Localization	10
		Steering angle	Motion Planning	10
			Motion Planning	40
Localization	20	Local position	Vehicle Interface	40
			Report Lane Position	50
			Motion Planning	33
			Object Classification	100
			Driving Behaviors	100
		Velocity state	Vehicle Interface	40
			Motion Planning	33
			Driving Behaviors	100
Road Detection: RLP	200	Lane position	Dyn. Obst. Pred.	100
			Driving Behaviors	100
			Motion Planning	100
Road Detection: DRAC	400	Drivable area coverage	Object Classification	100
			Motion Planning	200
Object Classification	100	Static obstacles	Motion Planning	100
		Dynamic obstacles	Dyn. Obst. Pred.	100
		Blind spot status	Driving Behaviors	500
Dyn. Obst. Pred	50	Dyn. obst. preds	Motion Planning	100
			Driving Behaviors	100
Route Planner	Variable	Planned route	Driving Behaviors	0
Driving Behaviors	91	Update RP	Route Planner	0
		CheckPT achieved	Route Planner	0
		Behavior profile	Motion Planning	100
Motion Planning: SL	200	Speed limit	MP - TS	0
Motion Planning: TS	80 ms typical, 450 ms max	Motion profile	Vehicle Interface	0
		TargetPT acheivability	Driving Behaviors	200

Table 3 Total reaction time in common situations

Critical dataflow	Fastest time(ms)	Slowest time(ms)
Response to a static obstacle		
OC	100	298
MP	80	159
VI	10	19
Total:	190	476
Response to a dynamic obstacle		
OC	100	298
DOP	50	198
MP-SL	200	399
MP-TS	80	159
VI	10	19
Total:	440	1073
Decision to procede at intersection		
OC	100	298
DOP	50	198
DB	91	280
MP-SL	200	399
MP-TS	80	159
VI	10	19
Total:	531	1353

The communications latency has the strongest impact on vehicle's reaction time to static and dynamic obstacles. The reaction time of Odin to a static obstacle, a dynamic obstacle, and intersection behavior is presented in Table 3 with results for best and worst case. The timing variability comes from the system of asynchronous loops. The best case assumes that each module receives the latest inputs just before the processing cycle begins. The worst case assumes that the latest inputs containing a newly sensed obstacle arrive just after the inputs are read, requiring processing to finish, and then run again to incorporate the new data. In practice, the expected latency should be in the middle of the best and worst cases. The latency from data transfer rates and the nondeterministic nature of Ethernet has been omitted as it is not a significant factor of latency in the system. An additional latency effect is that localization data could be as much as 99 ms old when it reaches Object Classification. This could displace sensed obstacles by as much as 0.99 m in the opposite direction of travel, when travelling at 10 m/s.

The software reaction time and other latency effects could be improved upon in the future, but were adequate for the Urban Challenge competition. Any error in the position of sensed obstacles due to the age of localization data reaching Object Classification is dependent on Odin's speed, and the error makes objects appear closer to Odin. This will not adversely affect safety because it will cause Odin to begin avoiding an obstacle earlier, and safety buffers in Motion Planning will prevent Odin from clipping the back of an obstacle. The reaction times to static and dynamic obstacles are only significant when new obstacles are sensed, or if data have been updated such as a change in position of a static obstacle or a deviation from prediction for a dynamic obstacle. The reaction times for these situations do not significantly hinder performance, especially when taking the sensing range of Odin into account. The reaction time when deciding to proceed at an intersection (stop sign or merging) was also sufficient for the Urban Challenge, but causes the vehicle extra delay when traveling and may prevent Odin from merging into a smaller vehicle gap. The Urban Challenge rules only penalize a vehicle for failing to merge into a vehicle gap of greater than 10 s, allowing Odin to still merge in time. If traffic is detected late and the increased reaction time for merging causes Odin to merge in front of traffic, the software can cancel a merge, preventing a collision.

C. Future Work

While the VictorTango software architecture performed extremely well in the DAPRA Urban Challenge, there exist areas for improvement that would enhance the performance as well as the portability of the architecture to new applications and domains. A large part of Odin's success can be attributed to the team's rigorous and exhaustive testing in both simulation and in the field. While the architecture supported this rapid design cycle development strategy, it would be desirable to incorporate machine learning into the process. Within Driving Behaviors, for example, individual behaviors and their position in the HSM 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. It would be beneficial to automatically generate or learn behaviors from experience [21,22]. Specifically, the use of machine learning and optimization techniques could be used to tweak control policies, state transitions, and command fusion parameters within behaviors. This would reduce the need for supervision and intervention by a human overseer in simulation testing and could reduce development man hours.

As shown in the discussion of software latency and message timing, there exist worst case scenarios that could cause decision making delays up to 1.35 s at an intersection. It has been shown that such delays were not enough to keep Odin from meeting the requirements set forth in the DARPA Urban Challenge. However, it would be advantageous to reduce these delays where possible. One solution would be to synchronize the messaging threads and data processing threads within each software module. Doing so would significantly reduce the gap between best-case and worst-case timing scenarios.

Another area for improvement lays in a more general set of error recovery procedures. Within the VictorTango architecture, error recovery of hardware and software happens on many different levels, from the Health Monitor to more specific solutions within individual modules. These error recovery procedures were developed through the testing process as problems arose. This sometimes led to an awkward sharing of error recovery responsibilities between modules. Furthermore, there is no guarantee that new errors or unforeseen situations are handled in a specific manner. Truly to expand the usability of the system to more varied and unstructured environments requires an error recovery system that remains distributed, but is more robust and adaptable to unforeseen events.

Lastly, while the decision to employ sensor-independent perception messages as virtual sensors enhances the portability of the software, the set of information they contained was minimal. It would be beneficial to add a small set of messages to communicate more metadata to the planning software. For example, Driving Behaviors operates using only the dynamic object list, and with no notion of the sensor modalities or sensor locations used in perception, it cannot accurately reason about occluded areas. While Driving Behaviors had mechanisms to deal with short periods of occlusion, a specialized Occlusion Hypothesizer with knowledge of sensor location and sensor state would make Odin more robust to longer periods of occlusion, such as the one in Fig. 17.

D. Conclusions

The DARPA Urban Challenge posed many important technical challenges to the mobile robotics community. Problems of perception, AI, navigation, and path planning all had to be addressed and implemented in reliable ways. Beyond these individual solutions, however, a fully autonomous system must bring these pieces together as parts of a much larger machine. Their interactions must be carefully designed and tested such that the entire system is capable of completing complex, multi-objective tasks in dynamic, unpredictable environments. Furthermore, to complete the Urban Challenge, this system must perform correctly 100% of the time, handling any errors or malfunctions internally and without any interaction with an external user.

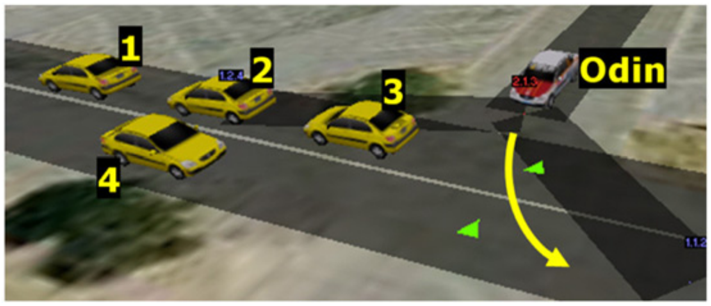

Fig. 17 Odin's view of vehicle 4 is temporarily occluded for a medium period of time.

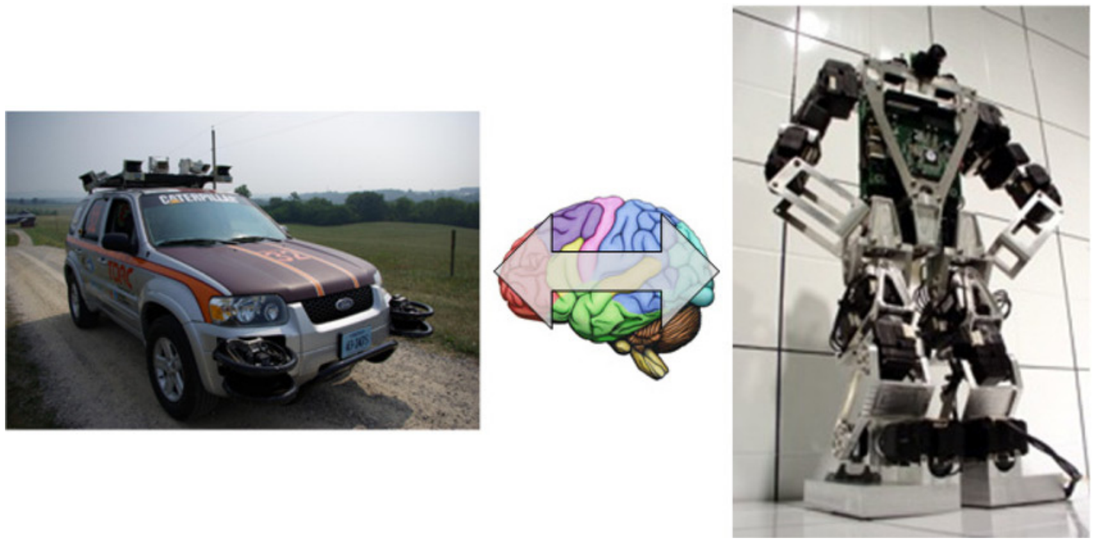

Fig. 18 Application of a portable software architecture across autonomous system domains.

To address this difficult problem, the VictorTango architecture employs a tri-level Hybrid Deliberative/Reactive approach that mimics biological intelligence in some aspects and takes advantage of brute force computing in others. This architecture exhibits a deliberative–reactive–deliberative progression, utilizing optimal planning techniques for high-level mission planning as well as low-level motion planning. Integration of a deliberative layer at the motion planning level is possible due to the development of an efficient, model-predictive search algorithm that takes advantage of pre-computed values and powerful processing capabilities. This addition subsequently shifts the typical scope and application of the behavior-based layer to higher-level decision making. A novel ASM is used within this reactive layer that takes advantage of both hierarchy and parallelism by placing a behavioral HSM for arbitration in sequence with a command fusion mechanism. As a result, contextual intelligence and emergent behavior are both preserved. Asynchronous perception processing modules continually fuse sensor data and produce a set of virtual sensors describing the environment. Virtual actuators are used to maintain hardware independence and to promote future reusability of software modules in new domains. Finally, the standardized JAUS communications protocol for inter-process messaging is used, allowing for dynamic reconfiguration of computing nodes, portability to new platforms, and the simple incorporation of simulation and hardware-in-the-loop (HIL) testing.

This software architecture has not only proven successful in the DARPA Urban Challenge, but it has also been ported to new vehicles and autonomous systems as illustrated in Fig. 18. By leaving room for modifications and improvements within software modules, the VictorTango architecture has proven adaptable to several different applications. One example of porting this approach to a drastically different domain has been on DARwIn [17,23], a bi-pedal, 2 foot tall humanoid robot capable of playing soccer. While a new set of experimental messages, behaviors, motion planning, and perception algorithms were developed, the fundamental problems of autonomy remain the same and the benefits of the VictorTango architecture still apply.

Acknowledgments

Many people and organizations contributed to the success of team VictorTango. The authors would first like to thank DARPA for creating and sponsoring the Grand Challenge series that motivated this work. Next, we would like to thank the other members of team VictorTango, especially the remaining graduate students: Tom Alberi, Dave Anderson, Aaron Dalton, Mike Webster, and David VanCovern as well as the long list of undergraduate students who contributed to this project. Finally, we would like to thank our sponsors, especially our premier sponsors: Caterpillar, Inc. and Ford Motor Co. This project would not have been possible without their support.

References

[1] Bacha, A., *et al.*, "Odin: Team VictorTango's Entry in the DARPA Urban Challenge," *Journal of Field Robotics*, Vol. 25, No. 8, 2008, pp. 467–492.
doi: 10.1002/rob.20248

[2] Urmson, C., *et al.*, "A Robust Approach to High-Speed Navigation for Unrehearsed Desert Terrain," *Journal of Field Robotics*, Vol. 23, No. 8, 2006, p. 467.
doi: 10.1002/rob.20126

[3] Murphy, R. R., *Introduction to AI Robotics*, MIT Press, Cambridge, MA, 2000.

[4] Russel, S., and Norvig, P., *Artificial Intelligence: A Modern Approach*, Pearson Education, Inc., Upper Saddle River, NJ, 2003.

[5] Brooks, R. A., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, 1986, pp. 14–23.

[6] Khatib, O., "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots." *The International Journal of Robotics Research*, Vol. 5, No. 1, 1986, pp. 90–98.
doi: 10.1177/027836498600500106

[7] Arkin, R. C., Riseman, E. M., and Hansen, A., "AuRA: An Architecture for Vision-Based Robot Navigation," *Proceedings of the DARPA Image Understanding Workshop*, Los Angeles, CA, 1997, pp. 413–417.

[8] Murphy, R., and Mali, A., "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, 1997, pp. 191–209.

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

[10] Simmons, R., Goodwin, R., Haigh, K., Koenig, S., and O'Sullivan, J., "A Layered Architecture for Office Delivery Robots," *Proceedings Autonomous Agents 97*, ACM Press, New York, NY, 1997, pp. 245–252.

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

[12] Thrun, S., *et al.*, "Stanley: The Robot That Won the DARPA Grand Challenge: Research Articles," *Journal of Field Robotics,* Vol. 23, No. 9, 2006, pp. 661–692.
doi: 10.1002/rob.20147

[13] Kelly, A. J., "A 3D State Space Formulation of a Navigation Kalman Filter for Autonomous Vehicles," CMU Robotics Institute Technical Report CMU-RI-TR-94-19, 1994.

[14] Webster, J., "A Localization Solution for an Autonomous Vehicle in an Urban Environment," Master's Thesis, Virginia Tech, Blacksburg, VA, 2007.

[15] Cacciola, S., "Fusion of Laser Range-Finding and Computer Vision Data for Traffic Detection by Autonomous Vehicles," Master's Thesis, Virginia Tech, Blacksburg, VA, 2007.

[16] Pirjanian, P., "Behavior Coordination Mechanisms: State-of-the-Art," Technical Report IRIS-99-375, Institute for Robotics and Intelligent Systems, University of Southern California, Los Angeles, CA, 1999.

[17] Hurdus, J. G., "A Portable Approach to High-Level Behavioral Programming for Complex Autonomous Robot Applications," Master's Thesis, Virginia Tech, Blacksburg, VA, 2008.

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

[19] Arkin, R. C., "Motor Schema based Navigation for a Mobile Robot: An Approach to Programming by Behavior," *IEEE International Conference on Robotics and Automation*, IEEE Publications, Piscataway, NJ, pp. 264–271, 1987.

[20] Bryson, J. J., "Hierarchy and Sequence vs. Full Parallelism in Reactive Action Selection Architectures," *From Animals to Animats 6(SAB00)*, MIT Press, Cambridge, MA, 2000, pp. 147–156.

[21] Alur, R., and Yannakakis, M., "Model Checking of Hierarchical State Machines," *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 3, 2001, pp. 273–303.
doi: 10.1145/503502.503503

[22] Argall, B., Browning, B., and Veloso, M., "Learning to Select State Machines using Expert Advice on an Autonomous Robot," *IEEE International Conference on Robotics and Automation*, IEEE Publications, Piscataway, NJ, pp. 2124–2129.

[23] Muecke, K., and Hong, D.W., "DARwIn's Evolution: Development of a Humanoid Robot," *IEEE International Conference on Intelligent Robotics and Systems*, IEEE Publications, Piscataway, NJ, 2007, pp. 2574–2575.

Christopher Rouff
Associate Editor