Sensing, Navigation and Reasoning Technologies for the DARPA Urban Challenge

Joel W. Burdick Noel duToit Andrew Howard Christian Looman
Jeremy Ma Richard M. Murray* Tichakorn Wongpiromsarn

California Institute of Technology/Jet Propulsion Laboratory

DARPA Urban Challenge Final Report
31 December 2007, Team Caltech

Abstract

This report describes Team Caltech's technical approach and results for the 2007 DARPA Urban Challenge. Our primary technical thrusts were in three areas: (1) mission and contingency management for autonomous systems; (2) distributed sensor fusion, mapping and situational awareness; and (3) optimization-based guidance, navigation and control. Our autonomous vehicle, Alice, demonstrated new capabiliites in each of these areas and drove approximate 300 autonomous miles in preparation for the race. The vehicle completed 2 of the 3 qualification tests, but did not ultimately qualify for the race due to poor performance in the merging tests at the National Qualifying Event.

1 Introduction and Overview

Team Caltech was formed in February of 2003 with the goal of designing a vehicle that could compete in the 2004 DARPA Grand Challenge. Our 2004 vehicle, Bob, completed the qualification course and traveled approximately 1.3 miles of the 142-mile 2004 course. In 2004-05, Team Caltech developed a new vehicle—Alice, shown in Figure 1—to participate in the 2005 DARPA Grand Challenge. Alice utilized a highly networked control system architecture to provide high performance, autonomous driving in unknown environments. The system successfully completed several runs in the National Qualifying Event, but encountered a combination of sensing and control issues in the Grand Challenge Event that led to a critical failure after traversing approximately 8 miles.

As part of the 2007 Urban Challenge, Team Caltech developed new technology for Alice in three key areas: (1) mission and contingency management for autonomous systems; (2) distributed sensor fusion, mapping and situational awareness; and (3) optimization-based guidance, navigation and control. This section provides a summary of the capabilities of our vehicle and describes the framework that we used the 2007 Urban Challenge.

*Corresponding author: murray@cds.caltech.edu

Figure 1: Alice, Team Caltech's entry in the 2007 Urban Challenge.

For the 2007 Urban Challenge, we built on the basic architecture that was deployed by Caltech in the 2005 race, but provided significant extensions and major additions that allowed operation in the more complicated (and uncertain) urban driving environment. Our primary approach in the desert competition was to construct an elevation map of the terrain sounding the vehicle and then convert this map into a cost function that could be used to plan a high speed path through the environment. A supervisory controller provided contingency management by identifying selected situations (such as loss of GPS or lack of forward progress) and implementing tactics to overcome these situations.

To allow driving in urban environments, several new challenges had to be addressed. Road location had to be determined based on lane and road features, static and moving obstacles must be avoided, and intersections must be successfully navigated. We chose a deliberative planning architecture, in which a representation of the environment was built up through sensor data and motion planning was done using this representation. A significant issue was the need to reason about traffic situations in which we interact with other vehicles or have inconsistent data about the local environment or traffic state.

The following technical accomplishments were achieved as part of this program:

1. A highly distributed, information-rich sensory system was developed that allowed real-time processing of large amounts of raw data to obtain information required for driving in urban environments. The distributed nature of our system allowed easy integration of new sensors, but required sensor fusion in both time and space across a distributed set of processes.
2. A hierarchical planner was developed for driving in urban environments that allowed complex interactions with other vehicles, including following, passing and queuing operations. A rail-based planner was used to allow rapid evaluation of maneuvers and choice of paths that optimized competing objectives while insuring safe operation in the presence of other vehicles and static obstacles.
3. A canonical software structure was developed for use in the planning stack to insure that contingencies could be handled and that the vehicle would continue to make forward progress

towards its goals for as long as possible. The combination of a directive/response mechanism for intermodule communication and fault-handling algorithms provide a rich set of behaviors in complex driving situations.

The features of our system were demonstrated in approximately 300 miles of testing performed in the months before the race, including the first known interaction between two autonomous vehicles (with MIT, in joint testing at the El Toro Marine Corps Air Station).

A number of shortfalls in our approach led to our vehicle being disqualified for the final race:

1. Inconsistencies in the methods by which obstacles were handled that led to incorrect behavior in situations with tight obstacles;

2. Inadequate testing of low-level feature extraction of stop lines and the corresponding fusion into the existing map;

3. Complex logic for handling intersections and obstacles that was difficult to modify and test in the qualification event.

Despite these limitations in our design, Alice was able to perform well on 2 out of the 3 test areas at the NQE, demonstrating the ability to handle a variety of complex traffic situations.

This report is organized as follows: Section 2 provides a high level overview of our approach and system architecture, including a description of some of the key infrastructure used throughout the problems. Sections 3–5 describes in the main software subsystems in more detail, including more detailed descriptions of the algorithms used for specific tasks. A description of the primary software modules used in the system is included. Section 6 provides a summary of the results from the site visit, testing leading up to the NQE and the team's performance in each of the NQE tests. Finally, Section 7 summarizes the main accomplishments of the project, captures lessons learned and describes potential transition opportunities. Appendix A provides a listing of additional modules that are referenced in the report along with a short description of the module.

2 System Overview

2.1 System Architecture

A key element of our system is the use of a networked control systems (NCS) architecture that we developed in the first two grand challenge competitions. Building on the open source *Spread* group communications protocol, we have developed a modular software architecture that provides inter-computer communications between sets of linked processes [1]. This approach allows the use of significant amounts of distributed computing for sensor processing and optimization-based planning, as well as providing a very flexible backbone for building autonomous systems and fault tolerant computing systems. This architecture also allows us to include new components in a flexible way, including modules that make use of planning and sensing modules from the Jet Propulsion Laboratory (JPL).

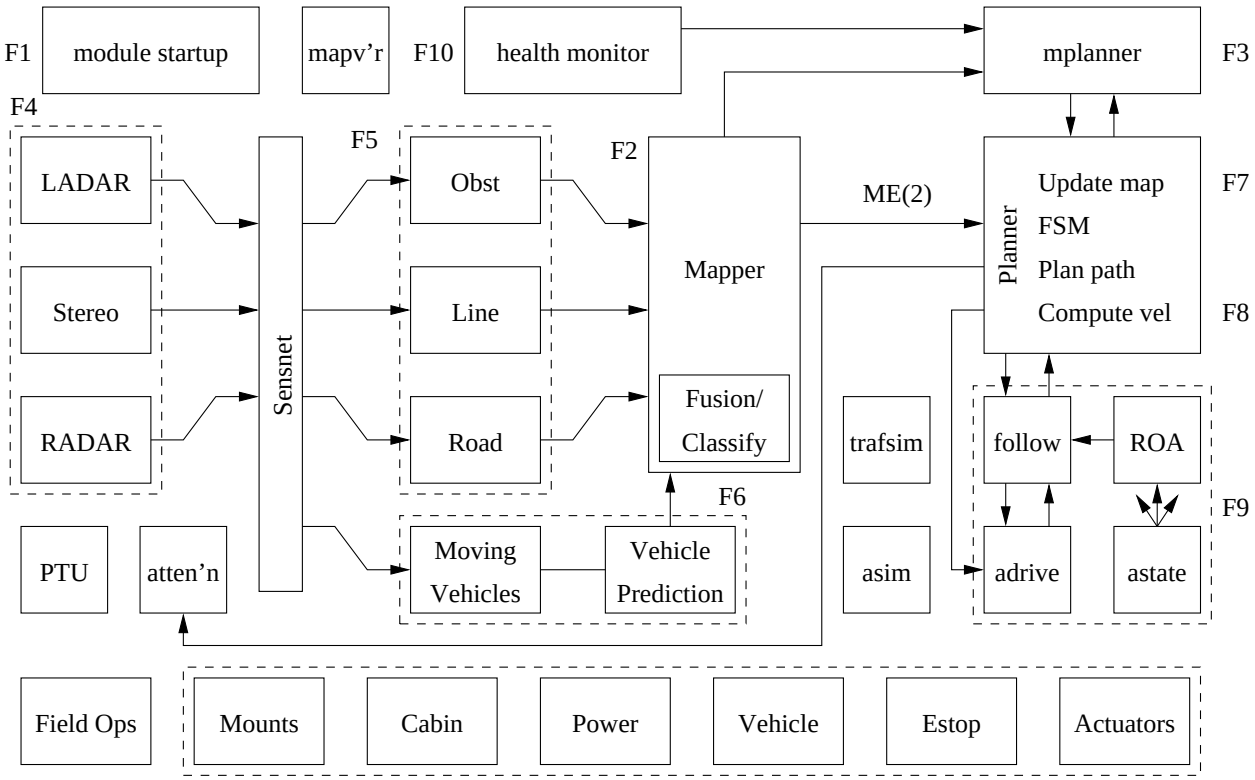


Figure 2: Systems architecture for operation of Alice in the 2007 Challenge. The sensing subsystem is responsible for building a representation of the local environment and passing this to the navigation subsystems, which computes and commands the motion of the vehicle. Additional functionality is provided for process and health management, along with data logging and simulation.

A schematic of the high-level system architecture that we developed for the Urban Challenge is shown in Figure 2. This architecture shares the same underlying approach as the software used for the 2005 Grand Challenge, but with three new elements:

Canonical Software Architecture for mission and contingency management. The complexity and dynamic nature of the urban driving problem make centralized goal and contingency management impractical. For the navigation functions of our system, we have developed a decentralized approach where each module only communicates with the modules directly above and below it in the hierarchy. Each module is capable of handling the faults in its own domain, and anything the module is unable to handle is propagated "up the chain" until the correct level has been reached to resolve the fault or conflict. This architecture is described in more detail in Section 5 and builds on previous work at JPL [2, 3, 7].

Mapping and Situational Awareness. The sensing subsystem is responsible for maintaining both a detailed geometric model of the vehicle's environment, as well as a higher level representation of the environment around the vehicle, including knowledge of moving obstacles and road features. It associates sensed data with prior information and broadcasts a structured representation of the environment to the navigation subsystem. The mapping module maintains a vectorized representa-

tion of static and dynamic sensed obstacles, as well as detected lane lines, stop lines and waypoints. The map uses a 2.5 dimensional representation where the world is projected into a flat 2D plane, but individual elements may have some non-zero height. Each sensed element is tracked over time and when multiple sensors overlap in field of view, the elements are fused to improve robustness to false positives as well as overall accuracy. These methods are described in more detail in Section 3.

Route, Traffic and Path Planning. The planning subsystem determines desired motion of the system, taking into account the current route network and mission goals, traffic patterns and driving rules and terrain features (including static obstacles). This subsystem is also responsible for predicting motion of moving obstacles, based on sensed data and road information, and for implementing defensive driving techniques. The planning problem is divided into three subproblems (route, traffic, and path planning) and implemented in separate modules. This decomposition was well-suited to implementation by a large development team since modules could be developed and tested using earlier revisions of the code base as well as using simulation environments. Additional details are provided in Section 4.

2.2 Project Modifications

The overall approach described in our original proposal and technical paper were maintained through the development cycle. After the site visit, the planning subsystem was modified due to problems in getting our original optimization-based software to run in real-time. Specifically, the NURBS-based dynamic planner described in the technical paper was replaced by a graph search-based planner. At a high level, these two planner both generated a path that obeyed the currently active traffic rules, avoided all static and dynamic obstacles, and optimized a cost function based on road features and the local environment. However, the rail-based planner separated the spatial (path) planning problem from the temporal (velocity) planning problem and made use of a partially pre-computed graph to allow a coarse plan to be developed very quickly. The revised planner is described in more detail in Section 4.2.

In addition, the internal structure of the planning stack was reorganized to streamline the processing of information and minimize the number of internal states of the planner. The probabilistic finite state machine used to estimate traffic state was replaced with a simpler finite state machine implementation.

Other differences from the technical paper include:

- The final sensor suite included two RADAR units mounted on separate pan-tilt units and no IR camera. This approach was used to allow long-range detection of vehicles at intersection (one RADAR was pointed in each direction down the road).
- Rather than using separate obstacles maps from individual LADARs and fusing them, a single algorithm that processed all LADAR data was developed. This approach improved robustness of the system, especially differentiating static and moving vehicles.
- The sensor fusion algorithms for certain objects were moved from the map object directly into the planner. This allowed better spatio-temporal fusion and persistence of intermittent objects.

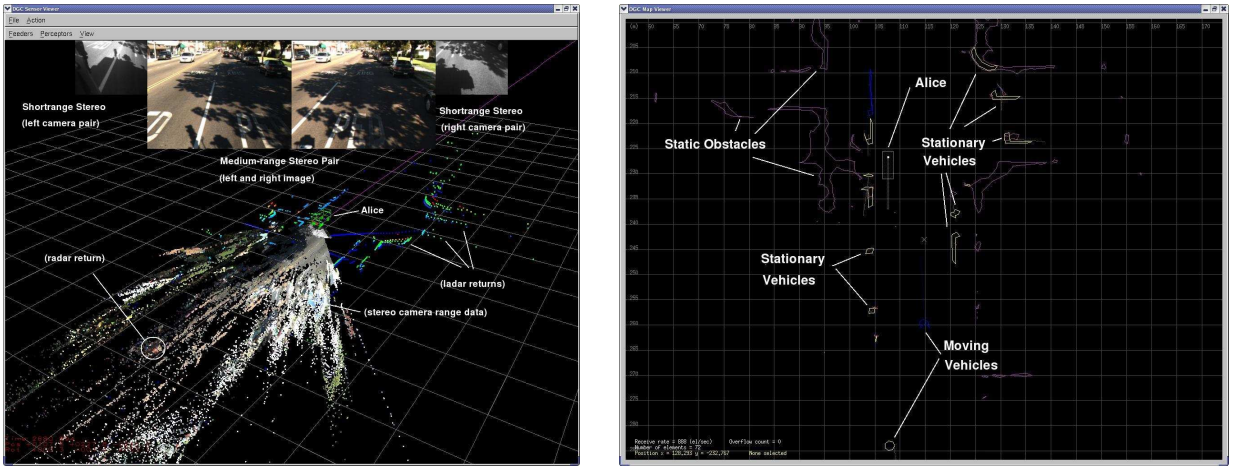


Figure 3: Screen shots from SensViewer (left) and MapViewer (right).

3 Sensing Subsystem

The sensing subsystem was developed from approaches for sensing, mapping and situational awareness that built off past work at Caltech/JPL. As can be seen from Figure 2, there are essentially three layers to the sensing subsystem. The flow of sensory data begins with the feeders in function block F4 and travels (via the SensNet interface) to the perceptors in function block F5. The perceptors then apply their respective perception algorithms and pass the perceived features to the map in function block F2, where fusion and classification is performed. In short, we can classify these layers as follows:

Sensing Hardware & Feeders: This layer consists of the low-level drivers and feeders that make the raw sensed data available to the perception algorithms. Each sensor is identified by a unique sensor-ID to keep things organized and allows the ease of incorporating new sensors as they get introduced.

Perception Software: This layer consists of the perception algorithms that take in the raw sensed data from the feeders and apply detection and tracking to extracted features from the data; such features include lines on the road, static obstacles, traversable hazards and moving vehicles.

Fused Map: This final layer consists of the vectorized representation of the world which we refer to as the "map". The map receives all the detected and tracked features that have been extracted at the perceptor level and fuses them to form a vectorized map database which is used by the planners.

Figure 3 contains screen shots illustrating the features of the sensing systems. The left figures shows the direct data from the various sensors represented in a 3D view. Images from the short and medium range cameras are shown at the top of the figure, and the LADAR, RADAR and stereo obstacle data are shown below. This information is processed by the perceptors, resulting in the fused data shown in the right figure. In this representation, obstacles are classified as static or moving. Lane data (not shown) is also stored in the map.

The following subsections address each layer of the sensing subsystem, with detailed descriptions of associated modules that were respectively used during the qualifying events at the NQE.

3.1 Sensing Hardware & Feeders

SensNet. Data from feeders is transmitted to perceptors using a specialized high-bandwidth, low-latency communication package called SensNet. SensNet's connection model is many-to-many, such that each feeder can supply multiple perceptors and each perceptor can subscribe to multiple feeders. Perceptors can therefore draw on any combination of sensors and/or sensor modalities to accomplish their task (e.g., a road perceptor can use both forward-facing cameras and forward-facing LADARs). SensNet will also choose an appropriate interprocess communication method based on the location of the communicating modules. For modules on the same physical machine, the method is shared memory; for modules on different machines, the method is Spread/TCP/Ethernet.

LADAR Feeder. The LADAR-feeder module works by interfacing with a variety of laser range finders using existing drivers which had been written (or rewritten) from previous races. For this particular race, the laser range finders we had used were the SICK LMS-221, the SICK LMS-291, and the RIEGL LMS-Q120. Depending on the sensor-ID given (specified as a command line argument to the module), this module calls the correct driver to interface with the desired scanner and broadcasts the resulting scan measurements across SensNet. The range scan is referenced in the sensor frame yet tagged with the vehicle state and appropriate matrix transforms to allow for flexibility in transforming the range points into any desired frame (i.e. sensor, local, vehicle).

Stereo Feeder. The stereo-feeder module works by reading in the raw images from all four cameras (two sets of stereo-camera pairs: one long-baseline and one medium-baseline). With the known baselines between cameras in a given stereo-camera pair, a disparity and range value is calculated for all corresponding pixels in both images using JPL stereo software. Finally, the raw image with disparity and range values is then sent across SensNet to all listening modules.

RADAR Feeder. The radar-feeder module works by interfacing with a TRW AC20 Autocruise RADAR unit and publishing the data to SensNet. The AC20 can report up to ten "targets" (intermittent, single-cycle returns) and ten "tracks"—internally tracked objects—at a refresh rate of 26 Hz. The internal tracking is fairly accurate, and when supplied with the vehicle's yaw rate and velocity, can compensate for its own motion. It filters out stationary returns, making it ideal for a car perceptor.

PTU Feeder. The Pan-Tilt-Unit (PTU) feeder is the controlling software that receives panning and tilting commands for one of two pan-tilt unit devices on the vehicle: the Amtec Robotics Powercube pan-tilt-unit or the Directed Perception PTU-D46-17. The panning and tilting commands can be specified in one of two ways:

- specifying pan-tilt angles in the PTU reference frame;
- specifying a point in the local-frame such that the line-of-site vector of the pan-tilt unit intersects that point (i.e. such that the pan-tilt-unit is looking at that point in space).

There are elements of this module that make it work like a feeder and elements that make it work like an actuator. The module is a feeder in the sense that it continually broadcasts its pan and tilt angles across SensNet. It is an actuator in the sense that it listens for messages that command a movement and executes those commands.

3.2 Perception Software

Line/Road Perceptor. There were essentially two line perceptors that were written for the DARPA Urban Challenge. The first line perceptor was identified as the "Line Perceptor" module and by an abuse of notation, the second line perceptor module was identified as the "Road Perceptor" module. A description of each module is provided below.

The Line Perceptor takes in raw sensory image data from the stereo camera feeders and applies a perception algorithm that filters out line features (e.g. stop lines, lane lines, and parking-lane lines) and sends them to the Map module. The details of the algorithm are outlined in the following steps:

1. The image is transformed into Inverse Perspective Mapping view (IPM) given the camera external calibration parameters. IPM works by assuming the road is a flat plane, and using the camera intrinsic (focal length and optical center) and extrinsic (pitch angle, yaw angle, and height above ground) parameters to take a top view of the road. This makes the width of any line uniform and independent of its position in the image, and only dependent on its real width in reality. It also removes the perspective effect, so that lines parallel to the optical axis will be parallel and vertical.
2. Spatial filtering is then done on the IPM image, using steerable filters, to detect horizontal lines (in case of stop lines) or vertical lines (in case of lanes). The filters are Gaussian in one direction and second derivative of Gaussian in the perpendicular direction, which is best suited to detect light line on dark background.
3. Thresholding is then performed on the image to extract the highest values from the filtered image. This is done be selecting the highest kth percentile of values from the image.
4. Line grouping is done on the thresholded image, using either of:

   - Hough transform: for detecting straight lines;
   - RANSAC lines: for detecting straight lines;
   - Histograms: a simplified version of Hough transform for detecting only horizontal and vertical lines;
   - RANSAC splines: for fitting splines to thresholded image.

   The Hough transform approach is the default mode of operation which provides flexibility in detecting lines of any orientation or position in the image. The orientations are searched between $\pm$ 10 degrees of horizontal or vertical, which allows for lane features that are not orthogonally aligned to the vehicle to be detected.

5. Validation and localization is then performed on detected lines and splines to better fit the image and to get end-points for the lines before finally sending to the Map module. We isolate the pixels belonging to the detected stop line using Bresenham's line drawing algorithm, and then convolve a smoothed version of the pixel values on the line with two kernels representing a rising and a falling edge, and getting the points of maximum response. The detected lines and splines are transformed back to the original image plane, and then to the local frame, which are then sent to the Map module.

The Road Perceptor module has a very similar architecture to the Line Perceptor module described above and can be briefly summarized in the following seven steps:

1. The first step loads images from the left camera taken from the stereo-feeder modules, which can either be from the middle baseline pair or the longer baseline pair. The default setting was to use the middle baseline pair.
2. The next step applies some pre-processing to enhance the loaded images (e.g. removing the hood of the vehicle from the bottom of the image and color separation for white vs. yellow lines).
3. Edge detection is next applied to the enhanced image which is done by applying an operator that extracts the main road hints in the form of edges or contours from the images.
4. Line extraction is then applied to the detected edges to extract candidate lines using the probabilistic Hough transform.
5. Line association then classifies the extracted lines as good lines or bad lines according to their color, geometrical properties, and relation with other lines.
6. Perspective Translation is then applied to the valid lines in much the same way described for the Line Perceptor module.
7. Line output is then executed on the translated lines, which are parameterized and broken into points then sent to the Map module.

Additional details on this algorithm are given in [6].

LADAR-Based Obstacle Perceptor. The LADAR-based Obstacle Perceptor has two main threads, preprocessing and tracking. The preprocessing thread is in charge of retrieving the latest laser range data (from all available LADARs) from the SensNet module and transforming it to the local frame. Once in the local frame, the data is then incorporated into a couple of occupancy grid maps:

- *Static Map* The static map is a map of free and occupied space represented by cells. All cells are first initialized to a negative value and when the actual returns are read in, corresponding cells are given a positive value. This map is used for grouping noisy obstacles like bushes and berms in an easy manner.
- *Road Map* The road map contains a map of large smooth surfaces likely to be road (stored as elevation data).

- *Groundstrike Map* The "groundstrike" map has cumulative information about what areas are likely to be LADAR scans generated by striking the ground. The data in this map is generated primarily from the elevation data gathered from the sweeping pan-tilt unit.

The second thread, the tracking thread, relies on discrete tracking of segments using a vectorized representation instead of a grid-based representation. For each LADAR, every incoming scan is segmented based on discontinuities (as a function of distance) and evaluated as a potential "groundstrike" based on the groundstrike probabilities of its constituent points. If a segment is accepted, it is associated with existing tracked segments or a new track is created. The function of the tracking is primarily to detect dynamic obstacles, since everything not dynamic is treated equivalently as static. Thus tracks are weighted by how likely they are to be cars (reflectivity, size, etc). The tracking just uses a standard Kalman filter for the centroid, though extra care is taken when initializing tracks to avoid misidentifying changing geometry as a high initial velocity. To combine the individual LADAR scans, each new set of scans is broken up into clusters of points (based on point-point distance), again to merge noisy obstacles into one large blob. For each cluster, the velocity is computed as the weighted average of each point's associated segment track, weighted by the status (confidence) of that track. Basic nearest obstacle association is done between scans; this is to maintain a consistent obstacle ID throughout the lifetime of a given obstacle. Dynamic obstacle classification occurs at this level, and is determined by the distance an obstacle has moved from its initial position, its velocity, and how long it has been visible. After a certain amount of time, an obstacle cannot be unclassified as dynamic. A reasonable geometry for each obstacle is then computed and the data is sent to the Map module.

LADAR-Based Curb Perceptor. The ladar-curb-perceptor (LCP) module is intended to detect and track the right side of the road. The "curb" in question need not be an engineered street curb, but rather denotes any sufficiently long, linear cluster of LADAR-visible obstacles that is nearly aligned with the vehicle's current direction of travel. This could be a row of bushes or sandbags, a berm, a building facade, or even a ditch. The LCP uses two sensors: the roof-mounted Riegl laser range finder (the beam which intersects the ground plane $\sim$15 m in front of Alice), and the middle front bumper SICK laser range finder (the sweep plane of which is parallel to the ground). The Riegl enables the LCP to see obstacles of any height, including negative, but only out to its ground-intersection distance. The SICK sees only obstacles over $\sim$1 m high, but out to a much greater range. Each scan of the LADARs is laterally filtered for step-edge detection; points that pass are considered obstacles which may be part of a curb (this procedure is robust to ground strikes due to pitching). Obstacle locations are converted to vehicle coordinates for aggregation over time, yielding a 2-D distribution as Alice moves forward. The current set of obstacles is clipped to a rectangular region of interest (ROI) about 30 m long along the direction of travel and several lanes wide, with its left edge aligned with Alice's centerline. From the entire set of obstacles, a subset of "nearest on the right" is extracted, one for each of thirty 1 m-deep strips orthogonal to the long axis of the ROI. A RANSAC line-fitting is performed on these nearest obstacles and the number of inliers is thresholded. If below threshold, the instantaneous decision is that no curb is detected; if above, the RANSAC-estimated line parameters are the instantaneous measurement of the curb location. Both the curb/no-curb decision and the curb line parameters are temporally filtered for smoothness.

RADAR-Based Obstacle Perceptor. The radar-obs-perceptor module is simply a wrapper around existing software that is embedded within the TRW-AC20 Autocruise RADAR. As described earlier the AC20 can report up to ten "targets" and ten "tracks"; however, only the tracks are used and sent to the Map module as the targets have been found to be quite noisy. Since both RADARs are attached to pan-tilt units, some pre-filtering of the tracks is required. This is necessary because if the base of the RADAR is moving with respect to the vehicle, the resulting tracks can be corrupted. To mitigate this, the RADAR-based obstacle perceptor subscribes to the associated PTU Feeder module through SensNet and marks detected tracks from the RADAR as void if the pan-tilt unit was found to have a non-zero velocity at the timestamp of the detected track. For those tracks that are not marked as void, they get packaged and stamped with an associated map element ID and sent to the Map module.

Stereo-Based Obstacle Perceptor. The stereo-obs-perceptor module uses disparity information provided by the stereo-feeder modules to detect generic obstacles. It uses a very simple algorithm, but works reasonably well. The following outlines the algorithm used for this perceptor:

1. Given the disparity image I_d, a buffer H is generated to accumulate votes from points with the same disparity occurring on a given image column (similar to a Hough Transform).
2. The accumulator buffer, H, is then searched for peaks higher than a threshold T_H. For each peak, it searches for the connected region of points above a lower threshold T_L containing the peak. One of the interesting features of this accumulator buffer approach is that it finds most of the vertical features since these will result in a peak, while flat features (like roads or lane lines) won't appear as peaks.
3. The next step is to fit a convex hull on the set of points found and transform the coordinates to local frame.
4. In this last step, each identified obstacle is tracked in time and a confidence value (probability of existence) is assigned/updated based on whether the track was associated with some measure or not. Only obstacles with a high confidence are sent to the mapper. The initial confidence is fairly low (0.4 in a scale from 0 to 1), so it takes a couple of frames before a new obstacle is sent to the map.

When implementing this algorithm, it became quite clear that a major hindrance in performance was due a bottleneck in the computing of the disparity of the stereo images. To account for this bottleneck and increase overall speed of this module, measures were taken to crop certain regions of the image that usually pertain to the sky or the hood of the vehicle. This reduced the search space of the disparity image and increased the cycle time by a few Hz. Other limitations to this algorithm were also identified and should be noted as well:

- Tracking and data association is very basic, but works well for static obstacles (i.e. no Kalman filter or other Bayesian filter).
- No velocity information is provided for the tracked obstacles.
- Sometimes, an obstacle is seen as two or more blobs, and sometimes two or more obstacles are seen as one. The tracking can deal with the first situation, but doesn't deal very well with the second, which is pretty uncommon though.

Attention. The Attention module is not so much a perceptor in that it does not *percept* any particular feature from a given data set. Instead, the Attention module interfaces directly with the Planning module and the PTU Feeder module to govern where the associated pan-tilt unit should be facing. The Attention module performs in the following manner:

1. We receive a directive from the Planning module about which waypoint the vehicle is planning to go to next and what the current "planning" mode is. The planning mode can be either:

   - INTERSECT LEFT - the vehicle is planning a left turn at an upcoming intersection
   - INTERSECT RIGHT - the vehicle is planning a right turn at an upcoming intersection
   - INTERSECT STRAIGHT - the vehicle is planning to go straight at the upcoming intersection
   - PASS LEFT - the vehicle is planning to pass left
   - PASS RIGHT - the vehicle is planning to pass right
   - U-TURN - the vehicle is planning a u-turn maneuver
   - NOMINAL - the vehicle is in a nominal driving mode

2. A grid-based "gist" map is next generated based on the received directives from the planning module. (The "gist" nomenclature comes from work developed in the visual attention community and is an abstract meaning of the scene referring to the semantic scene category.) The gist map details which cells in the grid-based map actually represent the "scene", whether the scene be an intersection, the adjacent lane or a stopped obstacle. For example, when making a right turn at an intersection, the gist of the scene would be all lanes associated with the intersection that can potentially turn into the desired lane.
3. A weighted cost is then applied to the gist map grid cells that is dependent on the planning mode. The weighting is chosen such that areas with high traffic flow and high potential of vehicle collision are given large weighting while those that are not, are given lower weighting. Using the above example for the right turn, the weighting of the cells associated with those lanes in the gist map would be chosen such that if any one lane did not have a stop line, a large weighting would be assigned; if all lanes had stop lines, a uniform weighting would then be applied.
4. A cost map is then generated which is initialized to the weighted gist map but keeps a memory of which cells have been *attended* (explained in the next step). Once a cell in the cost map has been visited, the cost of that cell is reduced but allowed to grow at a rate dependent on a specified growth constant. This allows for the revisiting of heavily weighted cells while still allowing lesser weighted cells to be visited.
5. Once the cost map is generated, the peak value of the cost map is then determined as a coordinate point in the local frame and sent to the PTU Feeder module. The PTU Feeder module will then execute the necessary motions to *attend* to the desired location. While in motion, the Attention module is restricted from sending any additional attention commands

to prevent an overflow of pan-tilt commands which could cause a hardware failure of the unit.

6. Updating the cost map is the final step in the algorithm. The PTU pan and tilt angles are continually read in from SensNet and the corresponding line-of-site vector for the pan-tilt unit is calculated. The grid cells in the cost map that are found to intersect with the line-of-site vector are then reduced to a zero cost so that the peak-cell search will not select this already attended cell. However, the growth of the cell (as described earlier) will then begin and will grow up to the maximum cost specified by the weighted gist map.

With regard to the pan-tilt unit on the roof and in the special case of the NOMINAL planning mode (which is the most common mode where the vehicle is doing basic lane following), a gist map is not generated. Instead, a series of open-loop commands are sent to the PTU Feeder module governing the roof PTU to sweep the area in front of the vehicle. This behavior allows the LADAR scans from the LADAR range finder attached to the PTU to generate a terrain map that is then used to filter out "groundstrikes" in the LADAR-Based Obstacle Perceptor (described earlier).

3.3 Fused Map

Mapper. The map is structured as a database of *map elements* and implements a query based interface to this database. Map elements are used to represent the world around Alice. A fundamental design choice was to move away from a 3-dimensional world representation to a simpler, but less accurate 2.5-dimensional world representation. Map elements are defined in the MapElement class and form the basis for this representation. The map elements represent planar geometry but with a possible uniform nonzero height, and are used to represent lines in the road as well as obstacles and vehicles.

The Mapper module maintains the representation of the environment which is used by the planning stack for sensor based navigation. It receives data in the form of map elements from the various perceptors on a specific *channel* of communication and fuses that data with any prior data of the course route in Route Network Definition File (RNDF) format. It then outputs a reduced set of map elements on a different channel across the network.The use of *channels* for map element communication made it easy to isolate which map elements were sent by which perceptors. This often helped in identifying bugs in the software and isolating it to either the fusion side of the map or the perception side of the map. This design choice also allowed for multiple modules to maintain individual copies of the map, which proved extremely useful for visualization tools.

Sensor fusion between different perceptors is also performed in the mapper module. This allows objects reported by multiple perceptors to be reported as a single object. The sensor fusion algorithm is based on proximity of objects of the same type. Groundstrike filtering can also be done at this stage by "fusing" elevation data with obstacle data (so that obstacles in the same location as the ground plane are removed.) Additional logic is required with a moving vehicle object is in the same location as a static obstacle; in this case the moving object type takes precedence since some perceptors do not sense motion.

In the final race configuration, the Mapper module was eventually absorbed into the Planning module, where it ran as a separate thread. The purpose of this was twofold: (1) It vastly reduced

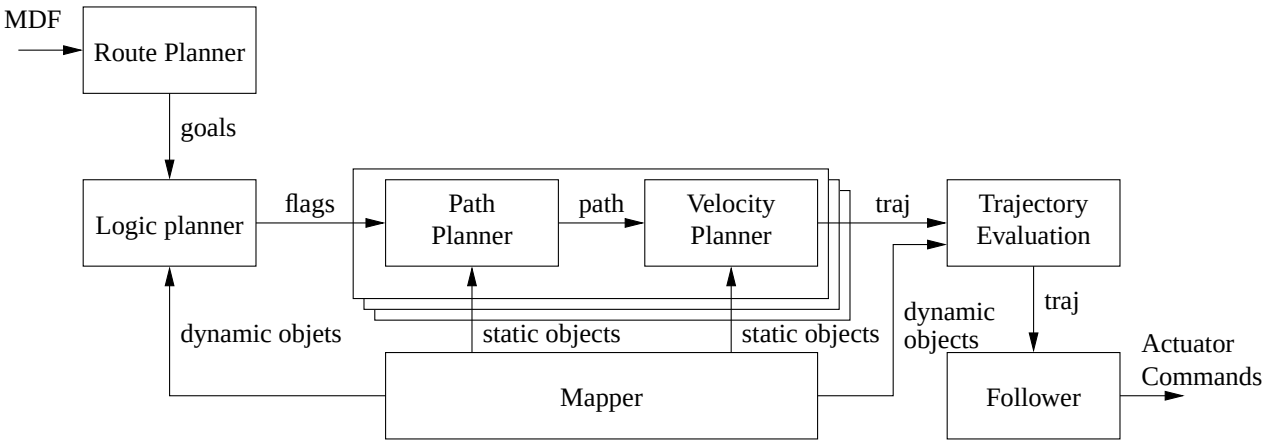


Figure 4: Three-layered planning approach: The mission-level planner takes the mission goal and specifies intermediate goals. These intermediate goals are passed to the tactical planner, which combined with the map information and the traffic rules, generates a trajectory. This trajectory is passed to the low-level planner, which converts the trajectory into actuator commands. The data-preprocessing step is also shown.

network traffic that was increased by the sending and receiving of thousands of map-elements when Mapper existed as it's own module. (2) It kept the only centralized copy of the map within the Planning module, where it was needed most.

4 Navigation Subsystem

The problem of planning for the Urban Challenge was complicated by three factors: first, Alice needed to operate in a dynamic environment with other vehicles. Not only was detection and tracking of these mobile objects necessary, but also their behavior was unknown and needed to be estimated and incorporated in the plan. Second, the requirement to obey traffic rules imposed specific behavior on Alice in specific situations. This meant that the Alice's situation (context) needed to be estimated and Alice had to act accordingly. However, since there were other vehicless on the course, Alice needed to be able to recover from situations where other vehicles did not behave as expected and thus adjust its own behavior. Lastly, Alice needed to be capable of planning in a very uncertain environment. Since the environment was not known a priori, Alice had to determine its own state, as well as the state of the world. Since this state cannot be measured directly, it needed to be estimated. This estimation process introduced uncertainty into the problem. Furthermore, the behavior of the dynamic obstacles is not known in advance, thus there was some uncertainty associated with their predicted future states. Another source of uncertainty is the fact that no model of a vehicle is perfect, and thus there is some process noise (i.e., given some action the outcome is not perfectly predictable).

The approach that Team Caltech followed in the planning was a three layer planning process, illustrated in Figure 4. At the highest level, the mission data is used to plan a route to the next checkpoint, as specified in the Mission Data File (MDF). This route is divided into intermediate goals, a subset of which is passed to the tactical level planner. The tactical planner is responsible

for taking this intermediate goal and the map (which is Alice's representation of the world), and designing a trajectory that satisfies all the constraints in the problem. These constraints include traffic rules, vehicle dynamics and constraints imposed by the world (obstacles, road, etc.). The trajectory is then passed to a low-level trajectory tracker. This feedback controller converts the trajectory into actuator commands that control the vehicle. There is also a data preprocessing step, which is responsible for converting the map into a format accessible to the planner, and a prediction step that estimates the future states of the mobile agents. These different pieces of the planning problem are described in this section.

A note on the coordinate system used is in order. For the planning problem, there are two frames of interest. The first coordinate system, called the world frame, is the geo-rectified frame (i.e., the frame that GPS data is returned in). This coordinate system is translated to waypoint 1.1.1 to make the coordinates more manageable. This is the coordinate system in which the tactical planning is conducted in since it is the coordinate frame used by DARPA in the Route Network Definition File (RNDF). The second coordinate frame of interest is the local frame. This frame is initialized to waypoint 1.1.1 as well, but is allowed to drift to account for state jumps. This is the frame that the sensors returned values in and is used in the low-level planning. The local frame ensures that obstacle positions are properly maintain relative to the vehicle, even if the GPS-reported state position jumps due to GPS-outages or changing satellite coverage.

4.1 Mission Level Planning

The mission-level planning has a number of functions. First, it is the interface between the mission management and health management systems. Second, it maintains a history of where we have driven before, which routes are temporarily blocked, etc. Third, it is responsible for converting the mission files into a set of intermediate goals and feeding these goals to the tactical planner as the mission progresses. The first function is discussed in Section 5 as part of the system-level mission and contingency management. The latter two functions are discussed here.

Route Planner. The route planner is the module that is responsible for finding a route given the RNDF and MDF. The RNDF is parsed into a graph structure, called the travGraph. The planner uses a Dijkstra algorithm to search this graph. Furthermore, the graph is used to store information about previous traverses of roads, including information about road blockages, etc. The route-planner is part of the mission-planner module, which encompassed these functions, as well as the interface with the mission and health management. The mission planner is implemented in the Canonical Software Architecture (CSA) and described in more detail in Section 5.1.

4.2 Trajectory Planning

The trajectory planning level is responsible for generating a trajectory based on the intermediate goals from the route planner, the map from the sensed data and the traffic rules. Before the planner can be executed, a number of preprocessing steps are necessary. First, the map data must to be converted into the appropriate data format used by the planner. Second, the future states of the detected mobile objects need to be determined. The planning approach followed here is known

as receding horizon control (RHC). In this approach, a plan is obtained that stretches from the current location to the goal. This plan is executed only for a short time and then revised at the next planning cycle, using an updated planning horizon.

The first step in the trajectory planning algorithm is to set up the planning problem. Traffic rules specify behaviors, and it is necessary to enforce these behaviors on Alice. The behavior currently required is determined via a finite state machine. This behavior included intersection handling and is implemented in the logic planner module. The behavior is enforced by setting up a specific planning problem. For example, the problem might not allow changing lanes in a region of the road where there is a double yellow line lane separator. The idea is to be able to solve multiple planning problems in parallel and then choose the best solution. This would have been useful when the estimate of the current situation cannot be obtained with sufficient certainty. This planning problem is then passed to the appropriate path planner.

Two types of path planning problems needed to be solved: planning in structured regions (such as roads, intersections, etc.) and unstructured regions (such as obstacle fields and parking lots). In our approach we used two distinct approaches for these problems, both of which are based on the receding horizon control approach.

For planning in structured regions, a graph is constructed offline, based on the RNDF. In this graph, the road geometry is inferred. The motivation behind this planning scheme is that the traffic rules imposed a lot of structure on the planning problem. This is one attempt to leverage this structure optimally. A second motivation is that, given that the graph defined the rail and lane changes and turns, it is possible to verify that we could complete a large portion of the course beforehand. The limitations of this approach are the assumed geometry of the road and potential state offsets. Given that we had aerial imagery of the test course, the first limitation is not overly constraining. Also, the planner had a mode that allowed it to switch to an “off-road” mode, where the planner is not constrained to the precomputed graph, but would navigated an obstacle field and try to reach a final pose. The second limitation is more worrisome, and it was decided to add multiple rails to each lane to allow the planner to choose the best rail, based on the detected road markings. This planner was implemented as the Rail Planner and is discussed some more below.

For path planning in unstructured regions, three parallel approaches were developed. The first approach is based on a probabilistic roadmap approach where a graph is constructed online. The approach followed is described below in the clothoid-planner section. The second approach, called the circle planner, constructed paths consisting of circular arcs and straight line segments. This approach was not actually used during the race. Both of these planners where spatial planners. The third approach is an optimal receding horizon control planner. This is a spatio-temporal planner (i.e., plans the trajectory directly). This planner, called the dynamic planner, was not used during the race, but is outlined in the sections below.

In order to plan in a dynamic environment, we separated the planning problem into a spatial planning problem and a spatio-temporal planning problem. This greatly simplifies the planning problem. Also, it is important to note that planning for dynamic obstacles vs. static obstacles is fundamentally different. For example, when following a car, one wants to plan where you want to drive and then adjust your velocity to obtain a safe trajectory. Thus, the separation of planning problems is justified. Also, in dealing with dynamic obstacles one did not necessarily want to

adjust your spatial path. There are some cases, however, where adjustment of the spatial path is required. For example, when passing an obstacle and there is a vehicle approaching from the rear in the lane we want to change into, it is not sufficient to only consider where that mobile object is currently, but we have to account for the future states of this mobile object. Similarly, when driving down a lane and there is a mobile object driving towards us, it is not sufficient to only adjust the velocity profile. This is accomplished by using the prediction information in two ways: first, to define regions prohibited to planning, and second to do a dynamic conflict analysis to determine possible collisions and avoid these early on.

The finer details of the modules used in the planning stack are given next.

Planner. The planner module functioned as the interface with the other mission-level planner and the map. The planner is implemented in the Canonical Software Architecture. It is responsible for maintaining a queue of intermediate goals, maintaining histories of some pervasive properties and sequencing the calls to the modules to solve the planning problem. In the case where multiple planning problems are set up, it would also maintain these different plans and select the best one (though this was not implemented). The planner module is also responsible for sending the trajectory that is obtained to the low-level planner for execution.

Since the planner has to interface with the different libraries, it was convenient to generate a module that maintained these interfaces. These interfaces are discussed next, before focusing on the functionality of the the individual library modules.

Planner Interfaces. The interfaces between the Planner module and the libraries needed to be maintained in a central location. These interfaces are maintained in a module called the temp-planner-interfaces, with the exception of the planners used for planning in unstructured regions. The reason for this separation was that the unstructured region planners used some objects that are slow to compile and this separation allowed a more efficient decomposition of the software.

Some of the interfaces defined in the temp-planner-interfaces module include the status data structure, which is used to report the status of the different libraries used in the planning problem. Also, the graph structure used for planning in the structured region is defined here, together with the path. Since the trajectory is an interface between the tactical- and low-level planners, this interface is defined elsewhere.

Logic Planner. The logic planner is responsible for high-level decision making in Alice. It has two functions: (1) to determine the current situation and come up with an appropriate planning problem to solve and (2) to do internal fault handling. These functions are not independent of each other, but we focus here on the the first function and discuss fault handling in more detail in Section 5.

The logic planner is implemented as two finite state machines. The first state machine is responsible for determining the current situation, by considering Alice's position in the world (e.g., proximity to intersections) and the status of the previous attempt at trajectory planning (i.e., if the planner failed due to blockage by an obstacle). These elements are factored in when setting up the planning problem to be solved in the current cycle. As an example of how this might work, consider a situation where Alice is on a two-lane, two-way road with a yellow divider. The initial problem is to drive down the lane to some goal location. This is given to the path planner to solve, but an obstacle blocks the lane. The path planner returns a status saying that it cannot solve the

problem and avoid obstacles (one of the constraints). In the next planning cycle, the logic planner can adjust the plan to now allow passing, at which point the planner will evaluate paths that move into the other lane.

The second state machine is used for intersection handling. Here we must account for the current map, the road geometry and Alice's position in the world to determine the correct intersection behavior. This behavior is then encoded in a planning problem, which is passed to the rest of the planning stack. A detailed desription of the intersection handling logic is available in [5].

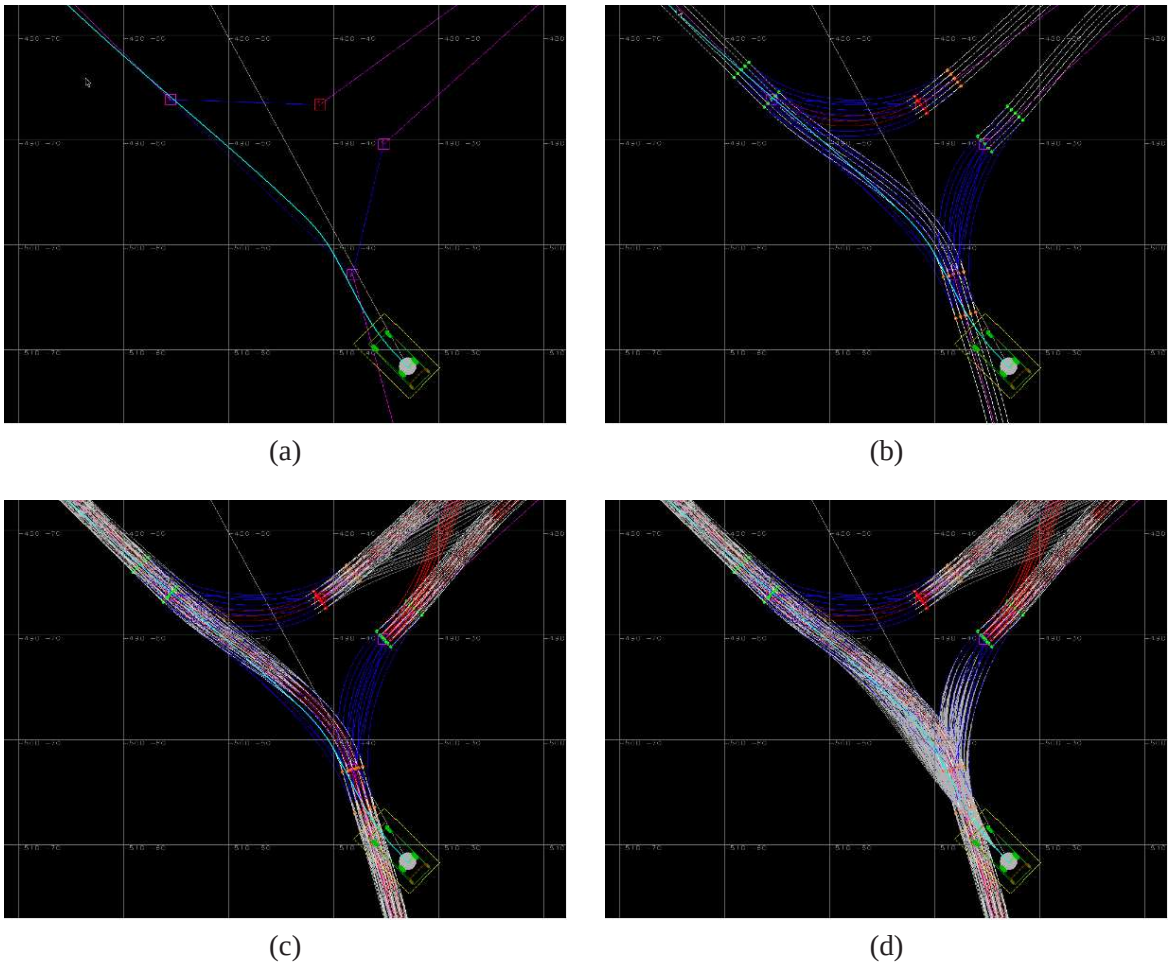
A note on dealing with uncertainty is in order at this point. The logic planner is susceptible to uncertainty in the current situation, as well as potentially uncertainty in the map. To overcome this hurdle, we had hoped to implement a probabilistic finite state machine. However, for this case it is conceivable that of the state transitions defined out of some state, none of these transitions are valid with high enough confidence. In this case, one approach would be to set up planning problems for the relevant transitions, solve the problems and evaluate the solutions. Unfortunately, this was never implemented due to lack of time.

Rail Planner. The rail planner's main function is to search over the precomputed graph to find the optimal path to the goal. Since this graph is defined in the world frame, the planner has to plan in the frame. The first step is to preprocess the map data. This data must be converted to either fields associated with the nodes of the graphs or weights associated with edges. Thus, the precomputed graph node locations are calculated and fixed offline, but the graph is updated online to reflect the latest sensing information. Once this step is completed, the optimal path to the goal can be calculated. To accomplish this, the planner uses an A* algorithm to search the graph. The cost function used in the optimization penalizes curvature, which is useful to avoid sharp maneuvers at high speed. Furthermore, the cost function tends to keep the vehicle in the center of the perceived lane. In addition, the obstacles are included in the cost generate plans that stay further away from obstacles when possible. In this way the uncertainty associated with the sensed objects is accounted for.

Figure 5 shows the different graphs created by the Rail Planner. The RNDF is first used to infer the geometry of the road and a single rail is placed down the (inferred) center of the lane (a). Turns through intersections are also defined. This is called the road graph. Since the road geometry is only approximately known, more rails are added to each lane to make the set of solutions to be searched less restrictive (b). Rail changes and lane changes are then defined on what is now the planning graph (c).

It was found that in some cases the precomputed graph was too constraining. This was because the rail change, lane changes and turns where precomputed. However, it was quite possible to have to deal with an obstacle between these predefined maneuvers. A function was implemented in this case to locally generate maneuvers (paths), called the vehicle-subgraph, that generated plans from the current location and connected to the precomputed graph as quickly as possible. This is shown in Figure 5d. This normally gives the planner enough flexibility to navigate these cases. The planning algorithm is also able to plan in reverse, when allowed, making the planner very capable.

As mentioned before, one of the concerns with using this planner is the inference of the road geometry from the RNDF. A mode of the planner was implemented where a local graph is generated online. This graph is much more elaborate that the vehicle subgraph discussed above and this



(a) (b)

(c) (d)

Figure 5: Operation of the rail planner.

capability allowed the planner to handle cases where the road did not line up with the expected geometry, including obstacle fields. The difficult problem became to determine when is it appropriate to switch into this mode. Unfortunately, this problem was never addressed and the segments for using this mode was hard coded base on manual inspection of the RNDF.

Clothoid Planner. The clothoid planner is the main planner used for planning in unstructured regions and is implemented in the s1planner module. This is a graph-search based planner, where the graph is generated online. The graph is constructed using a family of clothoid curves. Clothoid curves are curves with constant angular velocity and are commonly used for road layout design. The graph is constructed by expanding a tree of these families of curves. The tree is expanded until a relatively uniform covering of the state space is obtained. At this point, the graph is searched to find the optimal solution. A cost map is queried at each node (pose) to guide the search. At each pose considered, an obstacle overlap check is performed to ensure that the obstacles are avoided. Thus, obstacles are handled both as soft constraints, to push solutions away from obstacles, and as hard constraints. The output of this planner is a path in the same format as the rail planner.

Velocity Planner. The velocity planner accepts a spatial path and time parameterizes this path to obtain a trajectory. The velocity planner takes into account path features such as stop lines, obstacles on the path and obstacles close to the path. The planner considers the path, which has all the information necessary for velocity planning encoded in it, and specifies a desired velocity profile. For example, it will bring Alice to a stop at a desired deceleration and at a desired distance away from an obstacle. For obstacles on the side of the path, it will slow Alice down when passing close to these obstacles. Lastly, the velocity planner considers the curvature of the path and adjusts the velocities along the path accordingly. The velocity planner is compatible with the rail-, clothoid- and circle-planners. The output of the planner is a trajectory.

Prediction. Planning in an environment where the agents move at high speed requires some form of prediction of the future states of the mobile objects. Prediction of cars driving in urban environments is eased by the structure imposed on the environment, but is complicated by noisy sensory data and partial knowledge of the world state. The world state (map) and the mobile object's position in this world are necessary to determine behavior. Two approaches for prediction where investigated: (1) prediction based on particle filters and (2) prediction utilizing the structure in the environment and simple assumptions on the velocities of the mobile agents. The former approach was dropped since the data representation was not easily incorporated into the current planning approach. The latter approach has the disadvantage of not being of much use in unstructured regions.

The prediction information is used in two ways: first, the data is used to define restricted regions around mobile agents. This is especially useful when planning in intersections (such as merging) or planning lane changes. The second use is for dynamic conflict analysis. Here, the predicted future states of the mobile objects are compared to the planned trajectory of Alice. If a collision is predicted, an obstacle is placed in the map that alters Alice's plan and thus avoids a potential collision. Noisy measurements of the mobile object's state can cause prediction to sometimes be very conservative (when the velocity is off) or simply wrong (when the obstacle position in the partially known road network is estimated wrong).

4.3 Low-level Control and Vehicle Interface

The lower-level functions of the navigation system were accomplished by a set of tightly linked modules that controlled the motion of the vehicle along the desired path and broadcast the current state of the vehicle to other modules.

Follower. The follower module receives a trajectory data structure from planner and state information from astate. It sends actuation commands to gcdrive. Follower uses decoupled longitudinal and lateral PID controllers, to keep Alice on the trajectory. The lateral controller uses a nonlinear controller that accounts for limits on the steering rate and angle, and modifies its gains based on the speed of the vehicle [4].

Gcdrive. Gcdrive is the overall driving software for Alice. It works by receiving directives from follower over the network, checking the directives to determine if they can be executed and, if so, sending the appropriate commands to the actuators. Gcdrive also performs checking on the

state of the actuators, resets the actuators that fail, implements the estop functionality for Alice and broadcasts the actuator state. Also included in the role of gcdrive was the implemention of physical protections for the hardware to prevent the vehicle from hurting itself. This includes three functions: limiting the steering rate at low speeds, preventing shifting from occurring while the vehicle is moving, transitioning to the paused state in which the brakes are depressed and commands to any actuator except steering are rejected. (Steering commands are still accepted so that obstacle avoidance is still possible while being paused) when any of the critical actuators such as steering and brake fail.

Astate. The astate module was responsible for broadcasting the vehicle position (position, orientation, rates) data. This module read data from the Applanix hardware and processed the data to account for state jumps. It then broadcast the world and local frame coordinate for the vehicle.

Reactive Obstacle Avoidance. To ensure safe operation, it was decided to implement a low-level reactive obstacle avoidance (ROA) mechanism. This mechanism is the reason why the low-level planner needed to plan in the local frame. The ROA would evaluate LADAR data directly and when an object is detected within some box around Alice (which is velocity dependent), it would adjust the reference velocity of the trajectory to bring Alice to a stop in front of this object. One of the key issues that needed to be faced was making this mechanism sensitive enough to prevent collisions, but not so sensitive that it reacts to every false positive detection. Furthermore, the rest of the planner stack needed to be told that ROA is active (otherwise Alice stops for no apparent reason). Lastly, their needed to be a mechanism to override the ROA, otherwise there are situations where Alice would just be stuck indefinitely.

5 Mission and Contingency Management

Due to the complexity of the system and a wide range of environments in which the system must be able to operate, an unpredictable performance degradation of the system can quickly cause critical system failure. In a distributed system such as Alice, performance degradation of the system may result from changes in the environment, hardware and software failures, inconsistency in the states of different software modules, and faulty behaviors of a software module. To ensure vehicle safety and mission success, there is a need for the system to be able to properly detect and respond to unexpected events related to vehicle's operational capabilities.

Mission and contingency management is often achieved using a centralized approach where a central module communicates with nearly every software module in the system and directs each module sequentially through its various modes in order to recover from failures. As a result, this central module has so much functionality and responsibility and easily becomes unmanageable and error prone as the system gets more complicated. In fact, our failure in the 2005 Grand Challenge was mainly due to an inability of this central module to reason and respond properly to certain combination of faults in the system. This results from the difficulty in verifying this module due to its complexity.

The contingency management subsystem comprises the mission planner, the health monitor and the process control modules. The Canonical Software Architecture (CSA) was developed to

allow mission and contingency management to be achieved in a distributed manner. This function works in conjunction with the planning subsystem to dynamically replan in reaction to contingencies. The health monitor module actively monitors the health of the hardware and software to dynamically assess the vehicle's operational capabilities throughout the course of mission. It communicates directly with the mission planner module which replans the mission goals based on the current vehicle's capabilities. The process control module ensures that all the software modules run properly by listening to the heartbeat messages from all the modules. A heartbeat message includes the health status of the software. The process control restarts a software module that quits unexpectedly and a software module that identifies itself as unhealthy. The CSA ensures the consistency of the states of all the software modules in the planning subsystem. System faults are identified and replanning strategies are performed distributedly in the planning subsystem through the CSA. Together these mechanisms make the system capable of exhibiting a fail-ops/fail-safe and intelligent responses to a number different types of failures in the system.

5.1 Canonical Software Architecture

The modules that make up the planning system are responsible for reasoning at different levels of abstraction. Hence the planning system is decomposed into a hierarchical framework. To support this decomposition and separation of functionality while maintaining communication and contingency management, we implemented the planning subsystem in a canonical software architecture (CSA) as shown in Figure 6. This architecture builds on the state analysis framework developed at JPL [2] and takes the approach of clearly delineating state estimation and control determination. To prevent the modules from getting out of sync because of the inconsistency in state knowledge, we require that there is only one source of state knowledge although it may be captured in different abstractions for different modules.

A control module receives inputs and delivers outputs. The inputs consist of sensory reports (about the system state), status reports (about the status of other modules), directives/instructions (from other modules wishing to control this module), sensory requests (from other modules wishing to know about this modules estimate of the system state) and status requests (from other modules wishing to know about this module status). The outputs are the same type as the inputs, but in the reverse direction (reports of the system state from this module, status reports from this module, directives/instructions to other modules, etc).

For modularity, each module in the planning subsystem may be broken down into multiple CSA modules. A CSA module consists of three components—*Arbitration, Control* and *Tactics*—and communicates with its neighbors through directive and response messages, as shown in Figure 7. *Arbitration* is responsible for (1) managing the overall behavior of the module by issuing a merged directive, computed from all the received directives, to the *Control*; and (2) reporting failure, rejection, acceptance and completeness of a received directive to the *Control* of the issuing module. *Control* is responsible for (1) computing the output directives to the controlled module(s) based on the merged directive, received response and state information; and (2) reporting failure and completeness of a merged directive to the *Arbitration. Tactics* provides the core functionality of the module and is responsible for generating a control tactic or a contiguous series of control tactics, as requested by the *Control*.

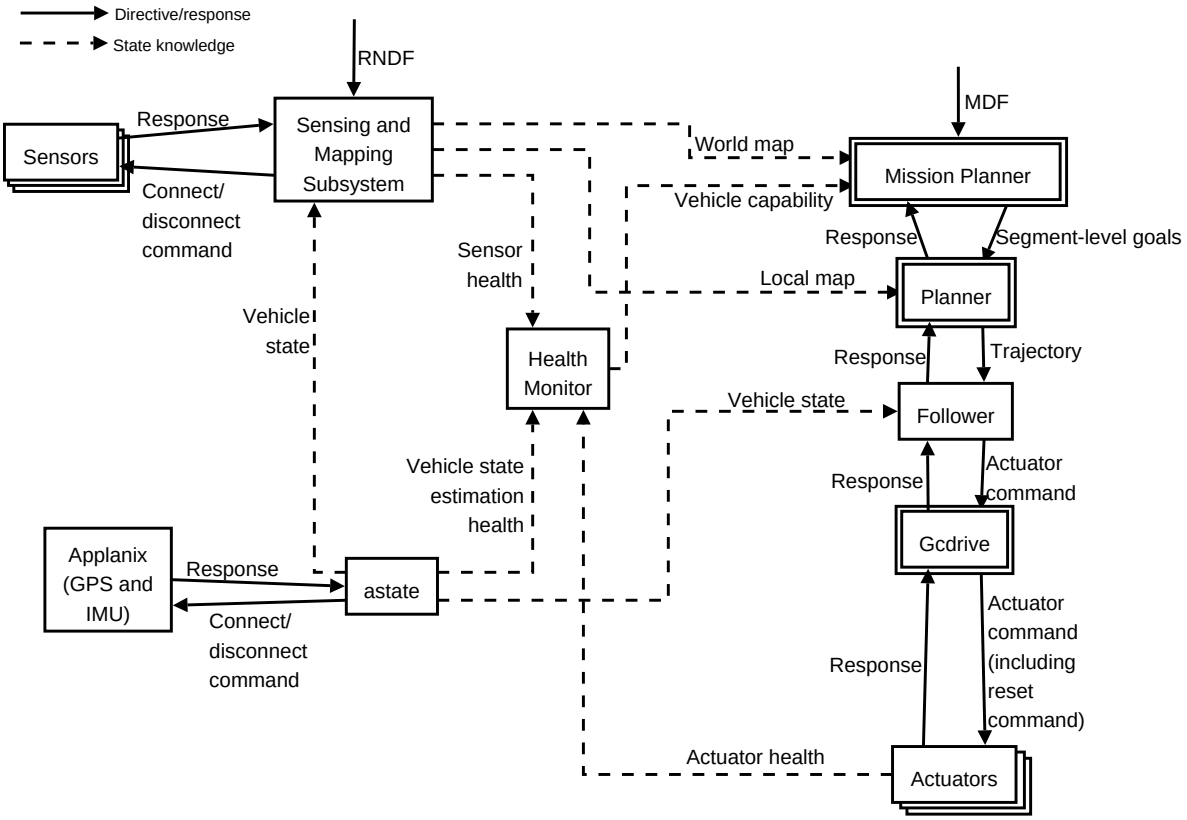


Figure 6: The planning subsystem in the Canonical Software Architecture. Boxes with double lined borders are subsystems that will be broken up into multiple CSA modules.

5.2 Health Monitor and Vehicle Capabilities

The health monitor module is an estimation module that continuously gathers the health of the software and the hardware of the vehicle (GPS, sensors and actuators) and abstracts the multitudes of information about these devices into a form usable for the mission planner. This form can most easily be thought of as vehicle capability. For example, we may start the race with perfect functionality, but somewhere along the line lose a right front LADAR. The intelligent choice in this situation would be to try to limit the number of left and straight turns we do at intersections and slow down the vehicle. Another example arises if the vehicle becomes unable to shift into reverse. In this case we would not like to purposely plan paths that require a U-turn.

From the health of the sensors and sensing modules, the health monitor estimates the sensing coverage. The information about sensing coverage and the health of the GPS unit and actuators allow the health monitor to determine the following vehicle capabilities: (1) turning right at intersection; (2) turning left at intersection; (3) going straight at intersection; (4) nominal driving forward; (5) stopping the vehicle; (6) making a U-turn that involves reverse; (7) zone region operation; and (8) navigation in new areas.

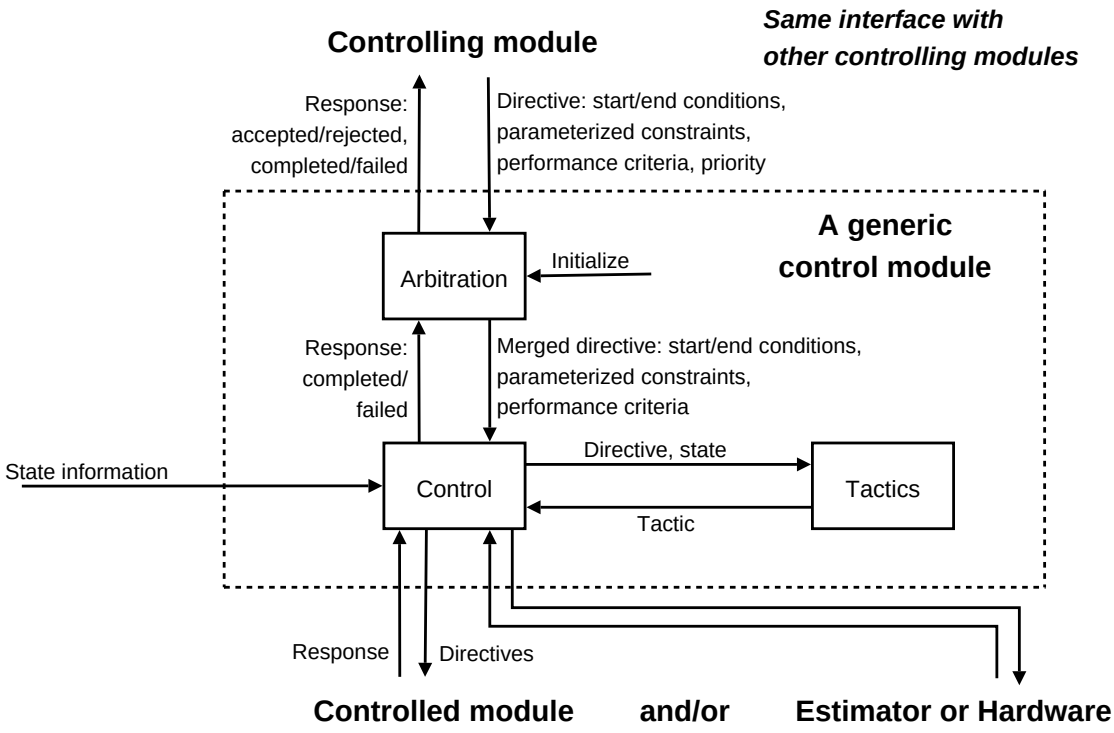


Figure 7: A generic control module in the Canonical Software Architecture.

5.3 Mission Planner

The mission planner module receives the vehicle capabilities from the health monitor module, the position of obstacles with respect to the RNDF from the mapper module and the MDF and sends the segment-level goals to the planner module. It has three main responsibilities and is broken up into one estimation and two CSA control modules.

Traversibility Graph Estimator. The traversibility graph estimator module estimates the traversibility graph which represents the connectivity of the route network. The traversibility graph is determined based on the vehicle capabilities and the position of the obstacles with respect to the RNDF. For example, if the capability for making a left or straight turn decreases due to the failure of the right front LADAR, the cost of the edges in the graph corresponding to making a left or straight turn will increase, and the route involving the less number of these maneuvers will be preferred. If the vehicle is not able to shift into reverse, the cost of the edges in the graph corresponding to making a U-turn will be removed.

Mission Control. The mission control module computes the mission goals that specify how Alice will satisfy the mission specified in the MDF and conditions under which we can safely continue the race. It also detects the lack of forward progress and replans the mission goals accordingly. The mission goals are computed based on the vehicle capabilities, the MDF, and the response from the route planner module. For example, if the nominal driving forward capability decreases, the mission control will decrease the allowable maximum speed which is specified in the mission

goals, and if this capability falls below certain value due to the failure in any critical component such as the GPS unit, the brake actuator or the steering actuator, the mission control will send a pause directive down the planning stack, causing the vehicle to stop.

Route Planner. The route planner module receives the mission goals from the mission control module and the traversibility graph from the traversibility graph estimator module. It determines the segment-level goals which include the initial and final conditions which specify the RNDF segment/zone Alice has to navigate and the constraints, represented by the type of segment (road, zone, off-road, intersection, U-turn, pause, backup, end of mission) which basically defines a set of traffic rules to be imposed during the execution of this segment-level goals, in order to satisfy the mission goals. The segment-level goals are transmitted to the planner module using the common CSA interface protocols. Thus, the route planner will be notified by the planner when a segment-level goal directive is rejected, accepted, completed or failed. For example, since one of the rules specified in a segment-level goal directive is to avoid obstacles, when a road is blocked, the directive will fail. Since the default behavior of the planner is to keep the vehicle in pause, the vehicle will stay in pause while the route planner replans the route. When the failure of a segment-level goal directive is received, the route planner will request an updated traversibility graph from the traversibility graph estimator module. Since this graph is built from the same map used by the planner, the obstacle that blocks the road will also show up in the traversibility graph, resulting in the removal of all the edges corresponding to going forward, leaving only the U-turn edges from the current position node. Thus, the new segment-level goal directive computed by the *Control* of the route planner will be making a U-turn and following all the U-turn rules. This directive will go down the planning hierarchy and get refined to the point where the corresponding actuators are commanded to make a legal U-turn.

5.4 Fault Handling in the Planning Subsystem

In our distributed mission and contingency management framework, fault handling is embedded into all the modules and their communication interfaces in the planning subsystem hierarchy through the CSA. Each module has a set of different control strategies which allow it to identify and resolve faults in its domain and certain types of failures propagated from below. If all the possible strategies fail, the failure will be propagated up the hierarchy along with the associated reason. The next module in the hierarchy will then attempt to resolve the failure. This approach allows each module to be isolated so it can be tested and verified much more fully for robustness.

Planner. The logic planner is the component that is responsible for fault handling inside the planner. Based on the error from the path planner, the velocity planner and the follower, the logic planner either tells the path planner to replan or reset, or specifies a different planning problem (or strategy) such as allowing passing or reversing, using the off-road path planner, or reducing the allowable minimum distance from obstacles. The logic for dealing with these failures can be described by a two-level finite state machine. First, the high-level state (road region, zone region, off-road, intersection, U-turn, failed and paused) is determined based on the directive from the mission planner and the current position with respect to the RNDF. The high-level state indicates the path planner (rail planner, clothoid planner, or off-road rail planner) to be used. Each of the

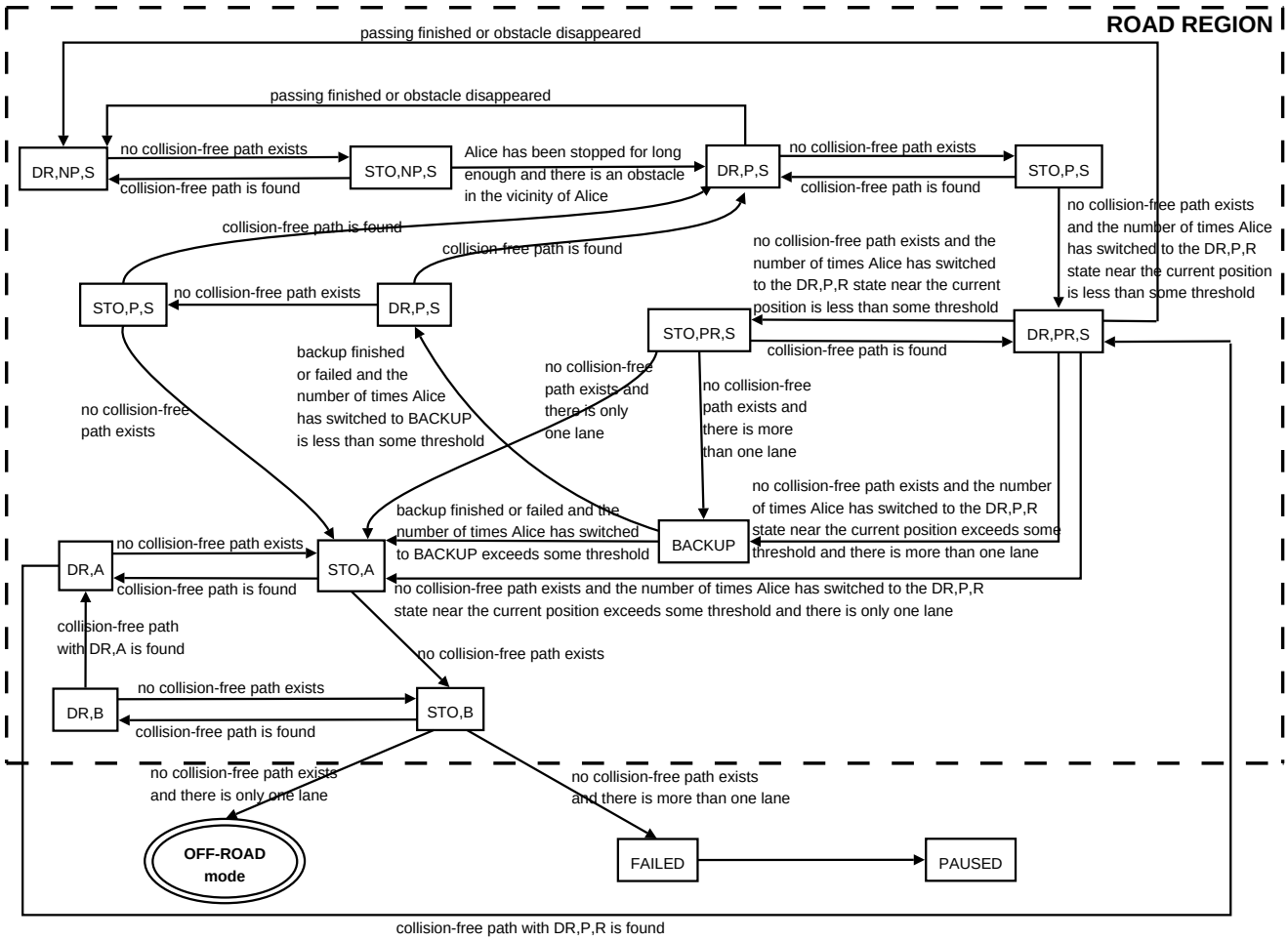


Figure 8: The logic planner finite state machine for the road region. Each state defines the drive state (DR $\equiv$ drive, BACKUP, and STO $\equiv$ stop when Alice is at the right distance from the closest obstacle as specified by the associated minimum allowable distance from obstacles), the allowable maneuvers (NP $\equiv$ no passing or reversing allowed, P $\equiv$ passing allowed but reversing not allowed, PR $\equiv$ both passing and reversing allowed), and the minimum allowable distance from obstacles (S $\equiv$ safety, A $\equiv$ aggressive, and B $\equiv$ bare).

high-level states can be further extended to the second-level state which completely specifies the planning problem described by the drive state, the allowable maneuvers, and the allowable distance from obstacles.

- *Road region* The logic planner transitions to the road region state when the type of segment specified by the mission planner is road. In this state, the rail planner is is the default path planner although the clothoid planner may be used if all the strategies involving using the rail planner fail. There are thirteen states and twenty seven transitions within the road region state as shown in Figure 8. The DR,NP state is considered to be the nominal state. The logic planner only transitions to other states due to obstacles blocking the desired lane or errors from the other planners.

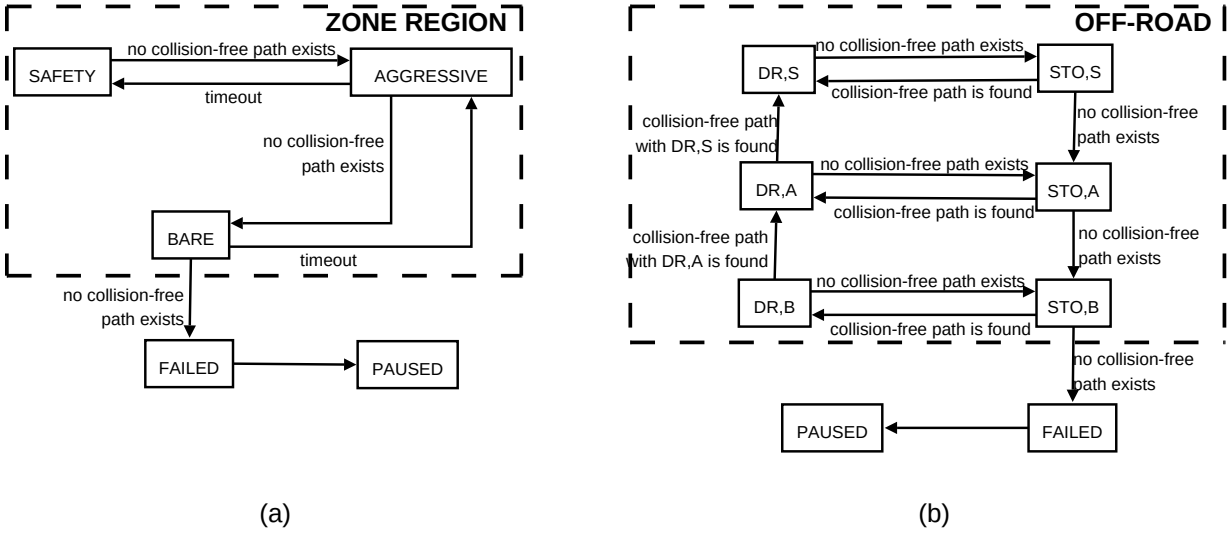


Figure 9: The logic planner finite state machine for the zone region (a) and off-road (b). Each state defines the drive state (DR $\equiv$ drive, and STO $\equiv$ stop when Alice is at the right distance from the closest obstacle as specified by the associated minimum allowable distance from obstacles) and the minimum allowable distance from obstacles (S $\equiv$ safety, A $\equiv$ aggressive, and B $\equiv$ bare).

- *Zone region* The logic planner transitions to the zone region state when the type of segment specified by the mission planner is zone. Reversing is allowed and since the clothoid planner is the default path planner for this state, the trajectory is planned such that Alice will stop at the right distance from the obstacle by default, so only three states and four transitions are necessary within the zone region state as shown in Figure 9(a).
- *Off-road* The logic planner transitions to the off-road state when the type of segment specified by the mission planner is off-road. Since passing and reversing are allowed by default, six states and ten transitions are necessary within the off-road state as shown in Figure 9(b).
- *Intersection* The logic planner transitions to the intersection state when Alice approaches an intersection. In this state, passing and reversing maneuvers are not allowed and the trajectory is planned such that Alice stops at the stop line. The rail planner is the default path planner. Once Alice is within a certain distance from the stop line and is stopped, the intersection handler, a finite state machine comprising five states (reset, waiting for precedence, waiting for merging, waiting for the intersection to clear, jammed intersection, and go), will be reset and start checking for precedence. The logic planner will transition out of the intersection state if Alice is too far from the stop line, when Alice has been stopped in this state for too long, or when the intersection handler transitions to the go or jammed intersection state. If the intersection is jammed, the logic planner will transition to the state where passing is allowed.
- *U-turn* The logic planner transitions to the U-turn state when the type of segment specified by the mission planner is U-turn. In this state, the default path planner is the clothoid planner. Once the U-turn is completed, the logic planner will transition to the paused state and wait

for the next command from the mission planner. If Alice fails to execute the U-turn due to an obstacle or a hardware failure, the logic planner will transition to the failed state and wait for the mission planner to replan.

- *Failed* The logic planner transitions to the failed state when all the strategies in the current high-level state have been tried. In this state, failure is reported to the mission planner along with the associated reason. The logic planner then resets itself and transitions to the paused state. The mission planner will then replan and send a new directive such as making a U-turn, switching to the off-road mode, or backing up in order to allow the route planner to change the route. As a result, the logic planner will transition to a different high-level state. These mechanisms ensure that Alice will keep moving as long as it is safe to do so.
- *Paused* The logic planner transitions to the paused state when it does not have any segment-level goals from the mission planner or when the type of segment specified by the mission planner is pause or end of mission. In this state, the logic planner is reset and the trajectory is planned such that Alice comes to a complete stop as soon as possible.

Follower. Although a reference trajectory computed by the planner is guaranteed to be collision-free, since Alice cannot track the trajectory perfectly, it may get too close or even collide with an obstacle if the tracking error is too large. To address this issue, we allow follower to request a replan from the planner through the CSA directive/response mechanism when the deviation from the reference trajectory is too large. In addition, we have implemented the reactive obstacle avoidance (ROA) component to deal with unexpected or pop-up obstacles. The ROA component takes the information directly from the perceptors (which can be noisy but faster) and can override the acceleration command if the projected position of Alice collides with an obstacle. The projection distance depends on the velocity of Alice. The follower will report failure to the planner if the ROA is triggered, in which case the logic planner can replan the trajectory or temporarily disable the ROA. We have also formally verified that through the use of the CSA, follower either has the right knowledge about the gear Alice is currently in even though it does not talk to the actuator directly and the sensor may fail; otherwise, it will send a full brake command to the gcdrive.

6 Results

Extensive testing on Alice was used to validate its capabilities and tune its performance. This section summarizes the major results of this testing.

6.1 Site Visit

The site visit consisted of four separate runs around a simple course consisting of a single intersection and a circular loop, as shown in Figure 10. After initial safety inspection and e-stop test, the first run consisted of driving around the loop once and was performed successfully.

The second run was a path planning run, in which a set of sparse waypoints were given and a route had to be planned that included performing U-turn operations in the stubs. On our first

Figure 10: Site visit course.

attempt at this run, the vehicle failed to perform the U-turn successfully, with an apparent loss of steering. A combination of high temperatures and a road surface that created large frictional forces with the tires caused a torque limit to be reached in the motor controller, resulting in a reset in the steering controller. This problem was remedied in a second attempt (after the fourth run) by reseting an internal parameters that lowered the commanded steering at slow speeds. After this change the test was performed successfully.

The third run involved driving multiple times around the loop with obstacles (stationary cars) placed at various points on the route. Alice detected and avoided all obstacles, and completed the run. For tests in which a vehicle was in the lane of travel, Alice signaled properly to move out of the lane and transitioned out of the lane at the required distances. Alice did not transition back into the lanes within the required distances, an artifact of the way in which the planning algorithm was implemented (there was an insufficiently high cost associated with gradually returning to the proper lane).

The fourth run focused on intersection operations. The run consisted of driving multiple times around the loop, with cars positioned at the intersection in different situations for each loop. Alice properly detected vehicles and respected the precedence order except for two occasion.

- In one instance, there were two cars queued up at the intersection opposite Alice. When Alice approached the intersection, it stopped, waiting a few seconds, and then continued through the intersection. According to the safety driver (who was in Alice), we had a small return coming up to the intersection and then the LADARs tilted down when we stopped. This caused the obstacles to disappear completely (the map subsystem had no memory at this point) and then reappear, so Alice decided that we were the first vehicle at the intersection.
- In the second instance, two vehicles were queued to the left of Alice. We stopped at the intersection and waited for the first vehicle. After that vehicle passed, we continued to wait at the intersection. After waiting for a while, DARPA motioned the second vehicle to go through and at that point Alice properly continued through the intersection. According to the

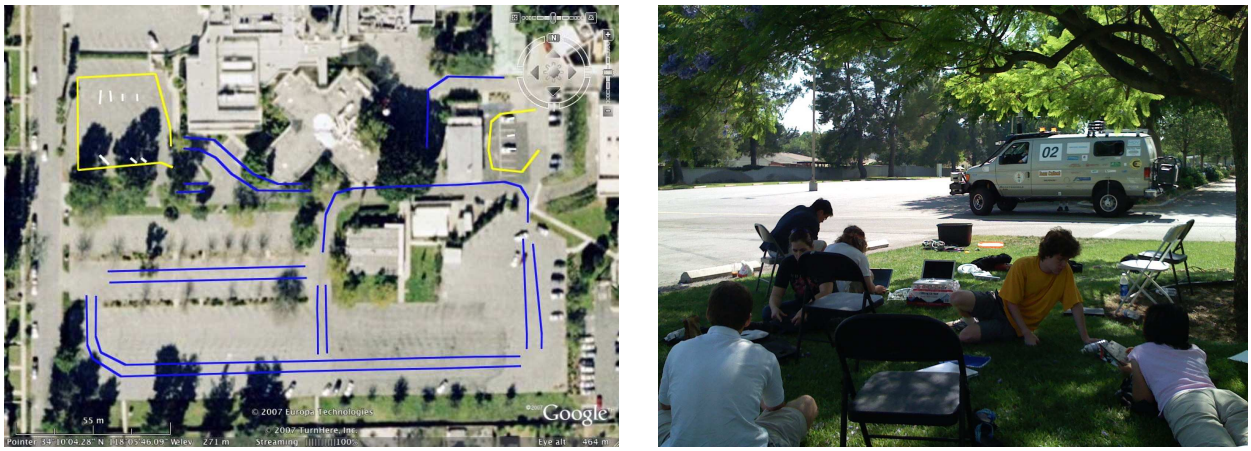


Figure 11: St. Luke Medical Center (Pasadena, CA)

internal logs, the second vehicle was partially in the opposing lane and that Alice interpreted this as a vehicle in the intersection, so it remained stopped.

While the site visit was executed more or less successfully, we identified several of limitations in the design. A major difficulty in preparing for the site visit was the brittleness of the finite state machine logic that accounted for traffic rules. Even with the limited complexity of the site visit tasks, the planner had dozens of states to account for different environmental conditions and driving modes. This made the planner very hard to debug. Some of the lower level control functions (including path following) were also found to be lower performance that we desired for the race. And finally, the accuracy and persistence of the sensed data need to be improvied.

The primary changes that were made after the site visit were (1) to simplify the traffic logic to use a very small number of modes; (2) to redesign the planning subsystem so that it made use of a graph-based planner instead of the originally proposed NURBS-based planner and (3) to streamline the planner software structure so that it acted as a single CSA module rather than separate modules for each internal function. These changes coincided with a decision to separate the path planning problem into a spatial planner (rail-planner) and a temporal planner (velocity-planner), rather than the originally planned spatio-temporal planner (dplanner). In addition, we rewrote the low-level control algorithms (follower) and implemented more robust functionality for detecting and tracking objects.

6.2 Summer Testing

During the summer of 2007, extensive testing and development was performed at two primary test sites: the former St. Luke Medical Center in Pasadena, CA and El Toro Marine Corps Air Station in Irvine, CA. Over the course of three months, approximately 300 miles of fully autonomous driving was performed in these locations.

Testing at the St. Luke Medical Center was performed in the (empty) parking lot of the facility, shown in Figure 11. While this area was quite small for testing, its proximity to Caltech allowed

Figure 12: El Toro Marine Core Air Station test area (Irvine, CA).

us to use the facility frequently and easily. A standard course was set up which could be used to verify the basic driving functionality and track performance. Some of the features of this course included tight turns, sparse waypoint areas, parking zones, overhanging buildings and trees, and tight maneuvering between structures.

El Toro Marine Corps Air Station was used for more extensive testing. This base is no longer in active use and was available for lease through a property management corporation. The primary RNDF used for testing is shown in Figure 12 This facility had all of the features specified in the DARPA Technical Criteria, including multiple types of intersections, multi-lane roads, parking zones, off-road sections, sparse waypoints, overhanging trees and tightly spaced buildings.

A total of 15 days of testing at El Toro were used to help tune the performance of the vehicle. The first long run with no manual interventions was a run of 11 miles on 19 September 2007, approximately 6 weeks before the race. The most number of miles driven in a single day was 40.5 miles on 16 October 2007. The highest average speed on a run of over 5 miles was 9.7 miles/hour on 16 October 2007. Additional testing included intersection testing with up to five vehicles, merging across traffic with cars coming from both directions, and defensive driving with traffic coming into the lane from a driveway and oncoming traffic driving in the incorrect lane.

6.3 National Qualifying Event

In this section we describe Team Caltech's performance in each of the three NQE test areas. We present each run in chronological order.

Run 1: Area B, Attempt 1. Area B consisted of tasks in basic navigation, including route planning, staying in lanes, parking and obstacle avoidance. An overview of Area B is shown in Figure 13. Basic navigation, stay in lane, parking. The MDF started in the starting chute area, then directed Alice to proceed down a road onto the main course. From there, the MDF direted Alice to drive down several different roads on the interiod of the course and eventually return to the starting area.

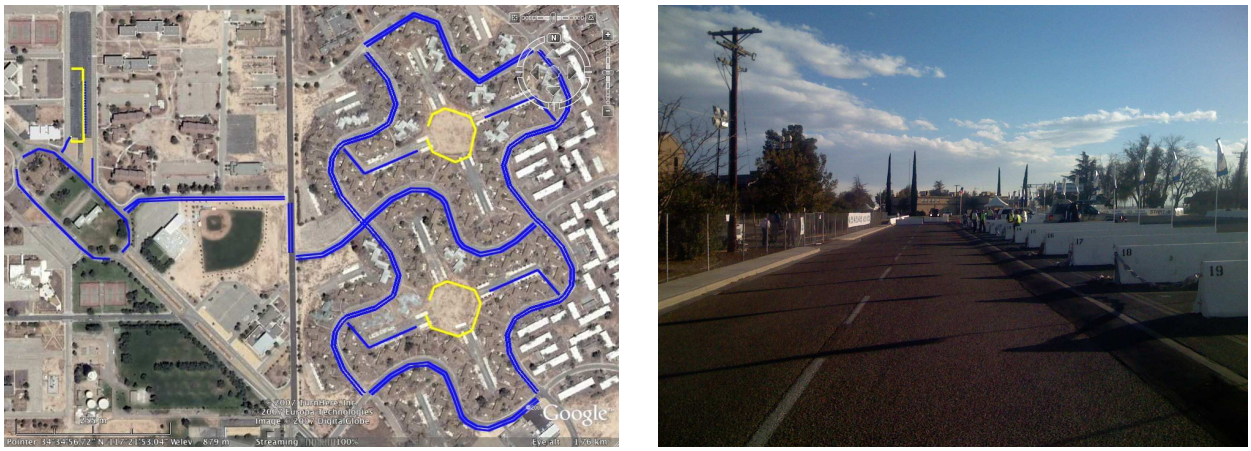


Figure 13: Test Area B

Alice encountered several difficulties on this run. First, the K-rails (concrete traffic barriers) in the startup chute were less than 1m away from Alice and the vehicle did not want to leave the chutes immediately. The same problem occurred at the exit of the startup area where K-rails formed a narrow gate. In order to proceed through the area, Alice had to progress through a series of internal planning failures before finally driving with reduced buffers on each side of the vehicle. After successfully leaving the area after about 5 minutes, Alice was performing well on the roads and entered the parking zone in the south part of the course. The spacing of the vehicles to each side of Alice was less than the required 1 meter buffer and Alice spent substantial time attempting to reorient itself to park in the spot. Once in the spot, Alice was unable to pull fully into the parking spot because the car in front of it was closer than the required 2 meter buffer. Alice was then manually repositioned and continued its run for a short period before the 30 minute time limit was reached.

As a result of this run, the code was changed to allow passing obstacles that are closer than 1m away from the vehicle. In addition, the tolerance of reaching waypoints in parking zones was relaxed.

Run 2: Area A: Attempt 1. This test consisted of merging into traffic with 10–12 manned vehicles circling around a "block", as shown in Figure 14. Vehicles are started in the center lane of the course and are supposed to make constant left turns, proceeding around the left loop of the course in the counterclockwise direction. Four vehicles with approximately equal spacing are circling around the larger loop in the counterclockwise direction. Six or more vehicles clustered together in groups of 1, 2 or 3 vehicles are circling the opposite direction. At the south intersection, Alice needs to merge into traffic after crossing one lane. At the north intersection Alice is supposed to make a left turn into oncoming traffic. The manned vehicles had a separation distance of 2 to 20 seconds. Therefore Alice had to sense a 10 second or longer gap and merge quickly into the gap.

In the first attempt of NQE run A, several bugs were uncovered. The first occured when Alice entered the intersection after determining that the path was clear. The proximity of a set of concrete barriers to the road meant that Alice could not complete the turn without coming close the the

Figure 14: Test Area A

barriers. The low-level reactive obstacle avoidance logic was using a different threshold for safe operation on the side of the vehicle (2 meters instead of 1 meter) and hence it would stop the vehicle partway through the intersection. This caused the intesection to become jammed (and generated lots of honks).

A second, related bug occurred in the logic planner that affected our wait at the intersection properly. While the intersection handler was active, another part of the higher-level logic planner could switch into the STOPOBS state if it detected a nearby vehicle (e.g, one of the human-driven cars was predicted to collide with Alice or its buffer region). This change in state de-activated the intersection handler and could cause the vehicle to enter the intersection when the path became clear (without invoking the proper merge logic). Table 1 gives a detailed analysis of the operation at each intersection. While in some cases the intersection handler was just interrupted but called again, it was canceled completely in other cases. If it was canceled, prediction was also not active. In these cases it almost caused two accidents with manned vehicles. At the north intersection, the software bug did not occur as logic planner did not switch into STOPOBS. This can be explained by the fact that by the nature of this intersection, no vehicle was crossing in front of Alice. As a result, merging was clean in all 7 scenarios at the north intersection.

Fixing the issues that were uncovered during this test required extensive changes at the NQE. First, the logic for reactive obstacle avoidance had to be changed to use a different safety buffer length in the front of the vehicle versus the sides (consistent with the logic used by the planner). Secondly, a rather major restructuring of the logic planner was required to insure that it did not skip the intersection handling logic until it had actually cleared an intersection. The changes were difficult to test at the NQE, even with extensive use of the testing areas (where no live traffic was allowed) and simulation.

Run 3: Area C, Attempt 1. Area C was designed to test intersection precedence, route planning and U-turn capabilities. The RNDF consisted of two intersections connected by a set of three roads, as shown in Figure 15. The major task in NQE run C was the correct handling of intersections with vehicles having precedence and to perform a U-turn at a road block. At the start of the run, the

Table 1: Analysis of performance in Area A, Attempt 1.

	Location	Vehicles passed	Missed gaps	Time passed	Comments
#1	S	8	N/A	32.0s	Interrupted by STOPOBS; prediction active
#2	N	1	0	7.7s	Clean merge
#3	S	6	N/A	17.0s	Interrupted by STOPOBS; prediction active
#4	N	2	0	14.6s	Clean merge
#5	S	11	1	>60.0s	Canceled by STOPOBS; prediction not active. Almost hit vehicle
#6	N	0	0	21.0s	Stopped too far left; other vehicles stopped
#7	S	3	N/A	23.0s	Canceled by STOPOBS; prediction not active
#8	N	3	0	16.3s	Clean merge
#9	S	6	0	38.2s	Clean merge
#10	N	1	0	7.9s	Clean merge
#11	S	4	N/A	12.0s	Canceled by STOPOBS; prediction not activated. Almost hit vehicle
#12	N	0	0	9.4s	Stopped too far left; other vehicles stopped
#13	S	6	N/A	34.0s	Interrupted by STOPOBS; clean merge
#14	N	0	0	4.2s	Clean merge

Figure 15: Test Area C

inner road between the intersection is blocked and the other roads are opened. The vehicle is commanded to go in a loop between the two intersections. At each successive intersection, a more complicated scenario is established. On the final run, one of the outer paths is blocked, requiring the vehicle to replan and choose a different route.

Table 2 gives an analysis of Alice's performance. The columns of the table indicate the intersection that was encountered, the number of vehicles at the intersection that had precedence at the time Alice arrived, the number of vehicles detected by Alice, and the number of times visibility was occluded by another vehicle.

Alice gave precedence correctly at all 7 intersections. At intersection #2 it was by accident that

Table 2: Analysis of performance in Area C, Attempt 1.

	Loc	# Veh w/ prec	# Veh seen	Lost visibility	Dur.	Comments
#1	N	0	0	0	2.4s	Empty intersection. Correct execution
#2	S	1	1	N/A	N/A	Interrupted by steering fault. Correct execution
#3	N	2	2	0	25.8s	Correct execution
#4	S	2	2	1	25.8s	Correct execution. One vehicle was blocking Alice's view while passing through intersection
#5	N	3	3	1	34.8s	Correct execution. One vehicle was blocking Alice's view while passing through intersection
#6	S	1	1	2	25.8s	Following at intersection, then giving precedence. Correct execution.
#7	N	3	3	2	37.8s	Correct execution. One vehicle was blocking Alice's view while passing through intersection

the power steering problems occurred when the other vehicle had already passed the intersection. While Alice was stationary, a troque fault in the steering caused a lower-level module to pause Alice for safety reasons. This event also triggers planner to switch into the state PAUSE which stops the intersection handling algorithm. After the system started up again and the intersection handler was called, the intersection was clear and Alice passed the intersection. Otherwise Alice might have made wrong assumptions about the time of arrival of other vehicles. The results from this run also demonstrate that ID tracking, checking for lost IDs and checking for visibility are crucial to the correct execution of the precedence. Without those backup algorithms, Alice would have misinterpreted the precedence order at intersections or would have lost vehicles in its internal precedence list.

After the intersection tests, Alice had to demonstrate correct execution of U-turns in front of road blocks. A bug was introduced in implementing the changes from Area A that caused the mission planner to crash during certain U-turn operations. The process controller properly restarted the mission planner after the crash, but Alice lost information regarding which part of the road was blocked. It thus alternated between the two road blocks and the run could not be finished within the time out limit but was considered a successful clean run.

The bug that caused the mission planner to crash was fixed in response to the results from this run.

Run 4: Area B, Attempt 2. Despite fixing the problems near the starting chute based on the previous attempt in Area B, Alice still had difficulty initializing to the properly state when it was placed in the startup chute. Due to delays in the launch of the vehicle by DARPA, we were able to correct the logic in the chute and launch the vehicle correctly.

With the changes in the buffer region, Alice was able to traverse through the start area and onto the course with little difficulty. At one point toward the beginning of the run, the control vehicle paused Alice because it appeared to be headed toward a barrier. This appears to be due to a checkpoint that was close to a barrier and hence Alice was coming close to the barrier in order to cross over the checkpoint. Alice was put back into run mode and continued properly.

The remainder of the run was completed with only minor errors. Alice properly parked in

Table 3: Analysis of performance in Area A, Attempt 2.

	Time	Action	Failure	Comments
#1	14:14:42	Merging, S inters'n		Clean merging after 14.9 s
#2	14:15:05	Exit of inters'n	Alice stopped	Stopped because of close obstacles and prediction
#3	14:15:56	Left turn	Problems following tight left turn, hit curb	Path/follower problems
#4	14:16:21	Merging, N inters'n		Clean merging after 38.6s
#5	14:17:21	Stopping, S inters'n	Stop line problems	Didn't stop at stop line and drove into intersection. Paused by DARPA
#6	14:19:21	Merging, N inters'n		Clean merging after 20.1s
#7	14:20:01	Stopping, S inters'n	Stop line problems	Didn't stop at stop line and drove into intersection. Paused by DARPA
#8	14:21:04	Exit out of inters'n	Pulling into on coming lane	Prediction stopped Alice, but manned car performed evasive maneuver. Paused by DARPA
#9	14:22:28	End of run		

the parking lot (the cars on the sides had been removed) and proceeded through the "gaunltet", a stretch of road in which a variety of obstacles had been placed. It then continued driving down several streets and through the northern zone, which had an opening in the fence. At several points in the run Alice ran over the curb after turning at intersections. Alice completed the mission in about 23 minutes.

Run 5: Area A, Attempt 2. In the second attempt at Area A, Alice's logic had been updated to ensure that intersection handler would not be overwritten by changing into another state within the logic planner's state machine. Unfortunately, an unrelated set of bugs caused problems on the second attempt. Table 3 summarizes the major events on this run. The primary errors in this run consisted of properly detecting the stop lines, which appeared in the logs to jump around in a manner that had not been previously seen (either in testing or other NQE runs).

To understand what happened at the stop lines, a bit more detail is required. The following steps and conditions that are necessary to stop at stop lines:

- Path planner - Creates path to stay in lane and to follow turns
- Planner - Search for stop lines close to path and store stop line information within the path structure
- Logic planner - Computes distance between Alice and next stop line found within path structure
- Logic planner - Depending on distance to stop line, switch in state STOP INTERSECTION
- Velocity planner - Detects state STOP INTERSECTION and modify velocity plan

These steps are necessary as the (spatial) path planner does not take into account stop lines, but

instead relies on the velocity planner to bring the vehicle to a stop. Therefore the function to find the closest stop line is the critical part of the algorithm.

All modules communicate with the skynet framework. During the race all skynet messages were written into a log file. Therefore the complete run can be replayed. Watching this replay and checking the log files, it became obvious that the main problem happened in computing the distance between Alice and the next stop line. To find the closest stop line, Alice performs the following actions:

- Search for all RNDF stop lines within this rectangle
- Project found RNDF stop lines onto path and choose closest
- Query map to obtain sensed stop line position for this stop line
- Choose closest node within path, required for velocity planner

In previous runs, sensed stop lines were only stored for 5 cycles after they were not picked up anymore by the sensors. This threshold was increased during the NQE as stop lines could not be seen by sensors when Alice's body was hiding the stop lines. Having a longer time-to-live value, false-positives were stored longer in the map. At this time, wrong data association in the map lead to jumping stop lines. When the vehicle approached the stop line in lap #2 and lap #3 the data association was right while approaching the intersection. As it came closer to the real stop line, the mapper bug assigned a false-positive stop line that was 3.2 meters behind Alice. In this case, Alice is assumed to have passed the stop line and did not stop as the threshold for passing a stop line was set to 3.0 meters. In lap #3 the stop line was moved ahead so that Alice was aiming for a stop line that was in the middle of the intersection. There was no algorithm in place detecting sudden changes in stop line positions.

7 Accomplishments and Lessons

Although Alice di not qualify for the race in 2007, the development of an autonomous vehicle capable of driving in urban traffic was very educational and rewarding. In this section we document some of the lessons learned and contributions of the project.

7.1 Lessons Learned

Team Caltech's approach to the Urban Challenge built on our experience from 2005, in which a combination of low-level failures were not properly handled by the software and Alice drove over a concrete barrier. To help mitigate the chances of a similar failure in 2007, a substantial effort was placed on systems engineering and systems architecture. Unfortunately, bugs that were similar in nature to what we experienced in 2005 again caused critical failures (this time in the qualifying event). As in 2005, the failure occurred in a situation that was not well-reflected in our testing and preparations.

The root cause for the fragilities in our system was lack of time and experience required to develop the software required for the Urban Challenge. Our original schedule planned on having

a fully functional system two months prior to the race, allowing ample time for testing and tuning. In reality, this point of technical progress was only reached approximately 2 weeks before the race, which meant that we were not able to test the software in a wide enough variety of situations to uncover some of the bugs and performance issues.

At a high level, the software architecture that was developed appears to be capable of performing autonomous operations at the level required for urban driving. With the exception of errors in robustly detecting stop lines, the sensing subsystem performed well and was extremely capable. The planning subsystem was more brittle and the finite state machine used to control the overall functioning of the planner proved to be difficult to verify and modify.

7.2 Technical Contributions and Transitions

Technical Contributions. Through this contact, the following technical contributions have been accomplished:

New technologies for mission and contingency management - A directive/response based architecture was developed to provide the ability to reason about complex, uncertain, spatio-temporal environments and to make decisions that enable autonomous missions to be accomplished safely and efficiently, with ample contingency planning. Building on expertise in high confidence decision-making and autonomous mission management at JPL, algorithms were developed to control the vehicle's sensing, estimation, mapping, planning and control systems in complex and uncertain conditions, while also ensuring safe operations.

Distributed sensor fusion, mapping, and situational awareness - Building on Caltech and JPL experience in sensory-based navigation—including feature classification and tracking, moving obstacle detection and tracking, visual odometry, and sensory-based mapping and localization—we developed a multi-layer decomposition of our sensed environment so that different levels of navigation and contingency management algorithms could operate in parallel while providing highly robust and safe operation. These modules operated in a highly distributed computational architecture.

Real-time, optimization-based navigation - we developed an optimization-based approach to guidance, navigation and control (GNC) that allows our vehicle to plan and execute locally optimal paths using a sensor-driven description of its environment. This approach was able to handle such issues as moving vehicles, traffic laws and defensive driving.

Transitions. Through this activity, we have established a strong working relationship with the Space Technologies sector of Northrop Grumman, including testing of advanced algorithms for motion planning on Alice (outside of the DGC program). We are also in discussions with the Systems Integration sector of Northrop Grumman regarding their interest in developing autonomous vehicle technologies for airport operations.

In addition, Caltech is currently supported under a Multidisciplinary University Research Initiative (MURI) grant in "Specification, Design and Verification of Distributed Embedded Systems" which will make use of our 2007 Urban Challenge experience (and our experimental platform) to pursue research in formal verification methods for complex, autonomous systems such as Alice.

Acknowledgments. The research in this paper was supported bin part by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-06-C-0146, the California Institute of Technology, Big Dog Ventures, Northrop Grumman Corporation, Mohr Davidow Ventures and Applanix Inc.

The authors would also like to thank the following members of Team Caltech who contributed to the work described here: Daniel Alvarez, Mohamed Aly, Brandt Belson, Philipp Boettcher, Julia Braman, William David Carrillo, Vanessa Carson, Arthur Chang, Edward Chen, Steve Chien, Jay Conrod, Iain Cranston, Lars Cremean, Stefano Di Cairano, Josh Doubleday, Tom Duong, Luke Durant, Josh Feingold, Matthew Feldman, Tony & Sandie Fender, Nicholas Fette, Ken Fisher, Melvin Flores, Brent Goldman, Jessica Gonzalez, Scott Goodfriend, Sven Gowal, Steven Gray, Rob Grogan, Jerry He, Phillip Ho, Mitch Ingham, Nikhil Jain, Michael Kaye, Aditya Khosla, Magnus Linderoth, Laura Lindzey, Ghyrn Loveness, Justin McAllister, Joe McDonnell, Mark Milam, Russell Newman, Noele Norris, Josh Oreman, Kenny Oslund, Robbie Paolini, Jimmy Paulos, Humberto Pereira, Rich Petras, Sam Pfister, Christopher Rasmussen, Bob Rasumussen, Dominic Rizzo, Miles Robinson, Henrik Sandberg, Chris Schantz, Jeremy Schwartz, Kristian Soltesz, Chess Stetson, Sashko Stubailo, Tamas Szalay, Daniel Talancon, Daniele Tamino, Pete Trautman, David Trotz, Glenn Wagner, Yi Wang, Albert Wu, Francisco Zabala and Johnny Zhang.

References

[1] L. B. Cremean, T. B. Foote, J. H. Gillula, G. H. Hines, D. Kogan, K. L. Kriechbaum, J. C. Lamb, J. Leibs, L. Lindzey, C. E. Rasmussen, A. D. Stewart, J. W. Burdick, and R. M. Murray. Alice: An information-rich autonomous vehicle for high-speed desert navigation. *Journal of Field Robotics*, 23(9):777–810, 2006.

[2] D. Dvorak, R. D. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in jpl's mission data system. In *Proceedings of 2000 IEEE Aerospace Conference*, 2000.

[3] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada. Engineering complex embedded systems with state analysis and the mission data system. *J. Aerospace Computing, Information and Communication*, 2, 2005.

[4] M. Linderoth, K. Soltesz, and R. M. Murray. Nonlinear lateral control strategy for nonholonomic vehicles. In *Proc. American Control Conference*, 2008. Submitted.

[5] C. Looman. Handling of dynamic obstacles in autonomous vehicles. Master's thesis, Universität Stuttgart, 2007.

[6] H. Pereira. Road marking for an autonomous vehicle in dynamic environments. Technical report, Faculdade de Engenharia da Universidade do Porto, 2007.

[7] R. D. Rasmussen. Goal based fault tolerance for space systems using the mission data system. In *Proceedings of the 2001 IEEE Aerospace Conference*, 2001.

A Additional Software Modules

In addition to the software modules described in the main text, a number of other modules were used as part of our system. Those modules are briefly described here.

ASim. Asim is a dynamic simulator for Alice that replaces the astate module. It accepts comments from gcdrive, simuilates the dynamics of the vehicle (including wheel slippage), and broadcasts the current vehicle state in a format compatible with astate.

Cotk. CoTK (Console Tool Kit) is a very basic display toolkit for text consoles. Implemented as a very thin layer over ncurses.

Circle Planner. The circle planner was one of two backup planners for the unstructured regions. This planner also constructed a graph from a family of curves. The curves considered in this case was circular arcs and straight line segments. This was a feasibility planner, and did not incorporate cost from the cost map. It considered obstacles as hard constraints. The graph search was done with an A* algorithm. This planner was very fast, and produced dynamically feasible solutions, but the solutions looked rather crude due to the family of curves used, which could easily have been remedied. This planner was tested but not used in the race.

DPlanner. An optimization-based planner was developed based on the use of NURBS basis functions combined with differential flatness, as described in the original proposal. This planner relied on a set of proprietary optimization algorithms that were developed by Northrop Grumman. The planner solves the complete spatio-temporal problem and is thus capable of accounting for the dynamic obstacles in the environment explicitly. The planner operated on a cost map, but also enforced hard constraints for obstacles. The solution obtained (a trajectory) satisfies the dynamics of the vehicle, as well as constraints on the inputs and state of Alice, while minimizing some cost function. The NURBS-based planner was not able to execute quickly enough to run in real-time, and so a rail-based planner was developed to replace it. The dplanner module was not used in the race.

MapViewer. A lightweight 2-D map and map object viewer built using FLTK. Mapviewer can be used to visualize map elements sent in and out of the mapper module.

RNDF-editor. A JAVA GUI program for editing RNDF files.

Skynet. The skynet library is used for group communications in Alice. It is a fairly thin wrapper around Spread. It supports broadcasting of messages to a given group name and subscribing to groups to receive relevant messages.

Sparrow. Sparrow is a collection of programs and a library of C functions intended to aid in the implementation of real-time controllers on Linux-based data acquisition and control systems. It contains functions for executing control algorithms at a fixed rate, communicating with hardware interface cards, and displaying data in real-time. For the DGC, the real-time data display was the primary usage.

Attachment: Intellectual Property

Team Caltech made use of the OTGX software package developed at Northrop Grumman, which was licensed to Caltech for use in the Urban Challenge. Based on our re-planning activities after the site visit, we dropped OTG from our system architecture and instead used the rail-planner software described in the final report (and provided to DARPA as a deliverable). Extensions of the OTGX algorithm funded through the DARPA contract (before the site visit) are provided to DARPA as part of the source code. Table 4 summarizes the intellectual property claims.

Table 4: Noncommercial intellectual property

IP Component	Basis for Assertion	Category	Organization
OTGX software	Developed exclusively at private expense	Restricted	Northrop Grumman