DISTRIBUTED MISSION AND CONTINGENCY MANAGEMENT FOR THE DARPA URBAN CHALLENGE

Tichakorn Wongpiromsarn and Richard M. Murray
Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA, USA
nok@caltech.edu, murray@cds.caltech.edu

Keywords: mission management, contingency planning, autonomous driving, fault handling, DARPA Urban Challenge

Abstract: We present an approach that allows mission and contingency management to be achieved in a distributed and dynamic manner without any central control over multiple software modules. This approach comprises two key elements—a mission management subsystem and a Canonical Software Architecture (CSA) for a planning subsystem. The mission management subsystem works in conjunction with the planning subsystem to dynamically replan in reaction to contingencies. 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 and the mission management subsystems through the CSA. The approach has been implemented and tested on Alice, an autonomous vehicle developed by the California Institute of Technology for the 2007 DARPA Urban Challenge.

1 INTRODUCTION

One of the major challenges in urban autonomous driving is the ability of the system 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. 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, an autonomous vehicle developed by the California Institute of Technology for the 2007 DARPA Urban Challenge, 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 these unexpected events which affect 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. Examples of such a central module are the behavior management module of the TerraMax Autonomous Vehicle (Braid et al., 2006), the supervisory controller (SuperCon) module of Alice previously developed for the 2005 DARPA Grand Challenge (Cremean et al., 2006) and the Fault Manager subsystem of the vehicle Theseus (Antonelli, 2003). A drawback of this approach is that the central module usually has so much functionality and responsibility and easily becomes unmanageable and error prone as the system gets more complicated. In fact, Team Caltech's failure in the 2005 DARPA Grand Challenge was mainly due to an inability of the SuperCon module to reason and respond properly to certain combination of faults in the system (Cremean et al., 2006). This resulted from the difficulty in verifying this module due to its complexity.

The complexity and dynamic nature of the urban driving problem make centralized mission and contingency management impractical. A mission management subsystem comprising the mission planner, the health monitor and the process control modules

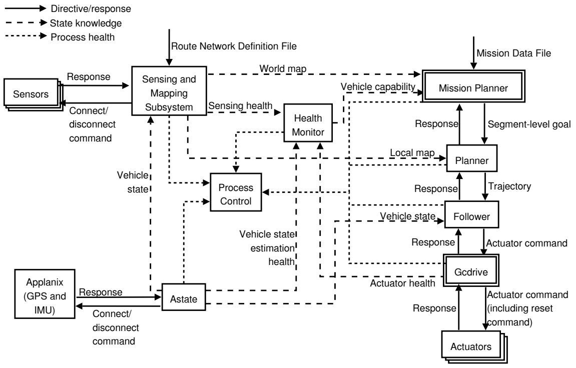

Figure 1: Alice's mission management and planning subsystems in the Canonical Software Architecture. Boxes with double lined borders are subsystems that will be broken up into multiple CSA modules.

and the Canonical Software Architecture (CSA) (Rasmussen and Ingham,) have therefore been developed to allow mission and contingency management to be achieved in a distributed manner. The mission management subsystem works in conjunction with the planning subsystem to dynamically replan in reaction to contingencies. As shown in Figure 1, the health monitor module actively monitors the health of the hardware and software components 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 monitoring the health of individual software modules and automatically restarting a software module that quits unexpectedly and a software module that identifies itself as unhealthy. An unhealthy hardware component is power-cycled by the software that communicates with it. 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 and the mission management subsystems 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.

Related work includes a holistic contingency management technology (Franke et al., 2006), a Mission Effectiveness and Safety Assessment (MENSA) technology (Franke et al., 2002), real-time fault detection and situational awareness (Dearden et al., 2004), the high level controller of the Intelligent Off-Road Navigator (Chen and Ümit Özgüner, 2006) and a model-based approach (Williams et al., 2003). These approaches rely on having a subsystem, similar to our mission management subsystem, capable of monitoring and assessing unexpected, mission-related events that affect the overall system operation and mission success. This subsystem may also be capable of suggesting a new strategy or operation mode for the planning subsystem or reconfiguring the system in response to these events. The CSA, however, attempts to facilitate these responsibilities of the mission management subsystem. By integrating the directive/response mechanism into the planning and the mission management subsystems, the mission management subsystem can assess most of the mission-related events by only reasoning at the level of failure or completion of its directives, thus eliminating the need to monitor or reason about the execution of the rest of the system and its behavior and interaction with the environment other than the health of the hardware and software components.

The contributions of this paper are: (1) a framework for integrating mission and contingency management into a planning system so that it can be achieved distributedly and dynamically; (2) a complete implementation on an autonomous vehicle system capable of operating in a complex and dynamic environment; and (3) an evaluation of the approach from extensive testing and some insight into future research directions. The remainder of this paper is organized as follows. Section 2 introduces the concept of the Canonical Software Architecture. Section 3 describes the mission management subsystem in more detail. Section 4 explains how system faults can be identified and handled distributedly through the CSA. Section 5 presents the results from the 2007 DARPA Urban Challenge's National Qualifying Event. Section 6 concludes the paper and discusses some future work.

2 CANONICAL SOFTWARE ARCHITECTURE

In many complex systems, the modules that make up the planning system are responsible for reasoning at different levels of abstraction. Hence, the planning system can be decomposed into a hierarchical framework. A Canonical Software Architecture has been developed to support this decomposition and separation of functionality, while maintaining communication and contingency management. This architecture builds on the state analysis framework developed at JPL and takes the approach of clearly delineating state estimation and control determination as described in

(Dvorak et al., 2000), (Rasmussen, 2001), (Barrett et al., 2004) and (Ingham et al., 2005). To prevent the inconsistency in the states of different modules due to the inconsistency in the state knowledge, we require that there is only one source of state knowledge although it may be captured in different abstractions for different modules.

In CSA, a control module receives a state and performs actions based on that state. The state contains estimates of the system status, the status of other modules relevant to this module, and the status of this module. Based on this state, an action computation is always made. Based on the actions taken, the state is updated.

A control module gets inputs and delivers outputs. The inputs consist of sensory reports (about the system state), status reports (about the status of other modules), directions/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), status requests (from other modules wishing to know about this module status) and direction requests (from other modules seeking direction from this module). 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, directions/instructions for other modules, etc).

For modularity, each software module in the planning subsystem may be broken down into multiple CSA modules. An example of the planning subsystem in CSA we have implemented on Alice is shown in Figure 1. 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 2. *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 completion of a received directive to the *Control* of the issuing module. Similar to the arbitration scheme of the subsumption architecture (Jones and Roth, 2004), the merged directive may simply be the received directive with the highest priority. Certainly, one can implement a more complicated arbitration scheme that involves dealing with multiple received directives simulaneously. *Control* is responsible for (1) computing the output directives to the controlled module(s) based on the merged directive, received responses and state information; and (2) reporting failure and completion 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*.

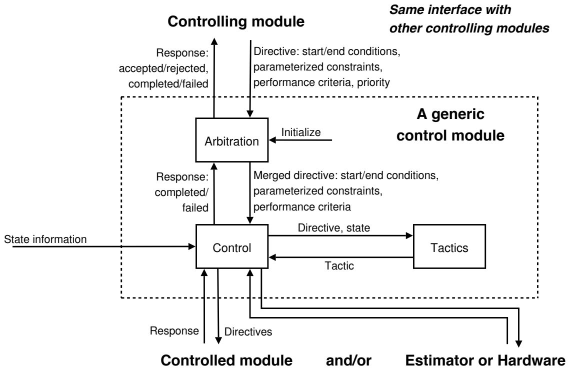
Controlling module
Same interface with other controlling modules
Response: accepted/rejected, completed/failed
Directive: start/end conditions, parameterized constraints, performance criteria, priority
A generic control module
Arbitration
Initialize
Response: completed/ failed
Merged directive: start/end conditions, parameterized constraints, performance criteria
State information
Control
Directive, state
Tactics
Tactic
Response
Directives
Controlled module
and/or
Estimator or Hardware

Figure 2: A generic control module in the Canonical Software Architecture.

3 MISSION MANAGEMENT SUBSYSTEM

The mission management subsystem is responsible for detecting changes in system health, assessing the impact of the changes on the vehicle's operational capabilities and adapt the mission plan accordingly in order to keep the vehicle operating safely and efficiently. This subsystem comprises the health monitor, the mission planner and the process control modules.

3.1 Health Monitor and Vehicle Capabilities

The health monitor module is an estimation module that continuously gathers the health of the software and hardware (GPS, sensors and actuators) components of the vehicle 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 mission with perfect functionality, but somewhere along the line lose a right front LIDAR. 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 driving in reverse; (7) zone region operation; and (8) navigation in new areas.

3.2 Mission Planner

The mission planner module receives the Mission Data File (MDF) that is loaded before each mission, the vehicle capabilities from the health monitor module, the position of obstacles from the mapper module and the status report from the planner module 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. For example, if the capability for making a left or straight turn decreases due to the failure of the right front LIDAR, 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 manuevers will be preferred. If the vehicle is not able to shift into reverse, the edges in the graph cornensponding to making a U-turn will be removed.

Mission Control The mission control module computes mission goals which specify how Alice will satisfy the mission specified in the MDF and conditions under which we can safely continue the mission. It also detects the lack of forward progress and replans the mission goals accordingly. 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 a failure of 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 determinates a sequence of segment-level goals to satisfy the mission goals. A segment-level goal includes 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 goal. 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.

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 planner is the main execution module for the planning subsystem. It accepts directives from the route planner component of the mission planner module and generates trajectories for Alice to follow.

The planner comprises four components—the logic planner, the path planner, the velocity planner and the prediction. The logic planner guides the vehicle at a high level by determining the current situation and coming up with an appropriate planning problem (or strategy) to solve. The path planner is responsible for finding a feasible path, subject to the constraints imposed by the planning problem. If such a path cannot be found, an error will be generated. Since Alice needs to operate in both structured and unstructured regions, we have developed three types of path planner to exploit the structure of the environment—the rail planner (for structured regions such as roads, intersections, etc), the off-road rail planner (for obstacle fields and sparse waypoint regions) and the clothoid planner (for parking lots and obstacle fields). All the maneuvers available to the rail planner are precomputed; thus, the rail planner may be too constraining. To avoid a situation where Alice gets stuck in a structured region (e.g. when there is an obstacle between the predefined maneuvers), the off-road rail planner or the clothoid planner may also be used in a structured region. This decision is made by the logic planner. The velocity planner takes the path from the path planner and planning problem from the logic planner and generates a time parameterized path, or trajectory. The prediction is responsible for predicting the future location and behavior of other vehicles.

The logic planner is the component that is responsible for fault handling inside the planner. Based on the error from the path planner and the follower, the logic planner either tells the path planner to replan or reset, or specifies a different planning problem 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 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 3. 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.
- **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 4(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 4(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 (Looman, 2007). 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-

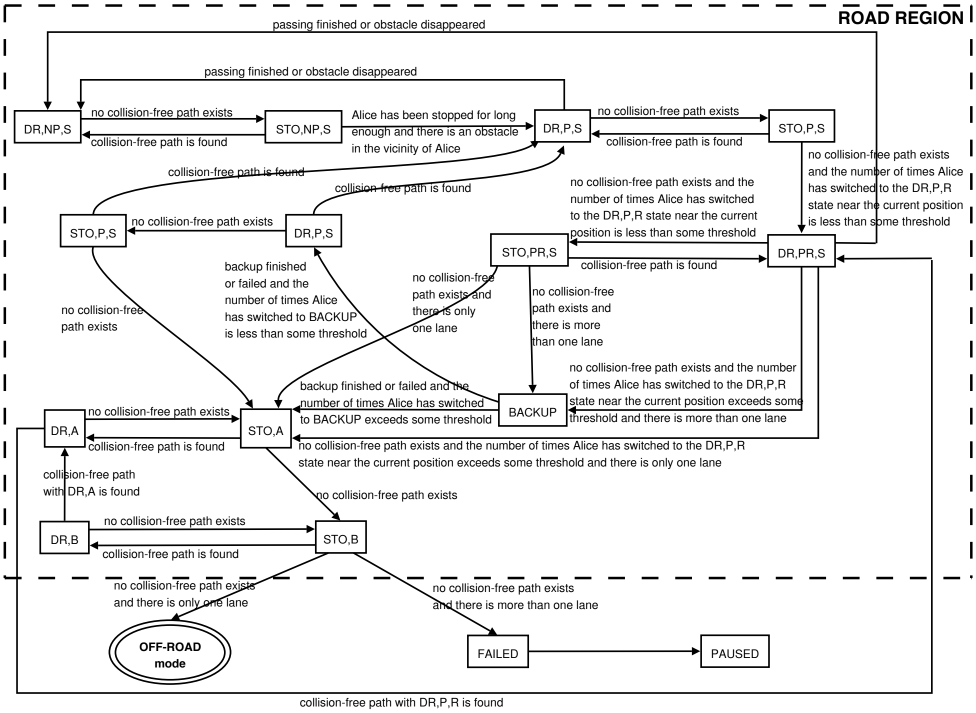
ROAD REGION
passing finished or obstacle disappeared
passing finished or obstacle disappeared
DR,NP,S
no collision-free path exists
collision-free path is found
STO,NP,S
Alice has been stopped for long enough and there is an obstacle in the vicinity of Alice
DR,P,S
no collision-free path exists
collision-free path is found
STO,P,S
no collision-free path exists and the number of times Alice has switched to the DR,P,R state near the current position is less than some threshold
collision-free path is found
collision-free path is found
STO,P,S
no collision-free path exists
DR,P,S
no collision-free path exists and the number of times Alice has switched to the DR,P,R state near the current position is less than some threshold
STO,PR,S
collision-free path is found
DR,PR,S
no collision-free path exists
backup finished or failed and the number of times Alice has switched to BACKUP is less than some threshold
no collision-free path exists and there is only one lane
no collision-free path exists and there is more than one lane
no collision-free path exists and the number of times Alice has switched to the DR,P,R state near the current position exceeds some threshold and there is more than one lane
backup finished or failed and the number of times Alice has switched to BACKUP exceeds some threshold
BACKUP
DR,A
no collision-free path exists
collision-free path is found
STO,A
no collision-free path exists and the number of times Alice has switched to the DR,P,R state near the current position exceeds some threshold and there is only one lane
collision-free path with DR,A is found
no collision-free path exists
DR,B
no collision-free path exists
collision-free path is found
STO,B
no collision-free path exists and there is only one lane
no collision-free path exists and there is more than one lane
OFF-ROAD mode
FAILED
PAUSED
collision-free path with DR,P,R is found

Figure 3: The logic planner finite state machine for the road region. Each state defines the drive state (DR ≡ drive, BACKUP, and STO ≡ 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 ≡ no passing or reversing allowed, P ≡ passing allowed but reversing not allowed, PR ≡ both passing and reversing allowed), and the minimum allowable distance from obstacles (S ≡ safety, A ≡ aggressive, and B ≡ bare).

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 The follower module receives a reference trajectory from the planner and vehicle state from the state estimator module and sends actuation commands to gcdrive. It uses decoupled longitudinal and lateral controllers to keep Alice on the trajectory (Linderoth et al., 2008). Although a reference trajectory computed by the planner is guaranteed to be collision-free, since Alice cannot track the trajectory perfectly, she 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, even though the follower does not talk to the actuator directly and the sensor may fail, it always either has the right knowledge about the gear Alice is currently in, or sends a full brake command to the gcdrive in

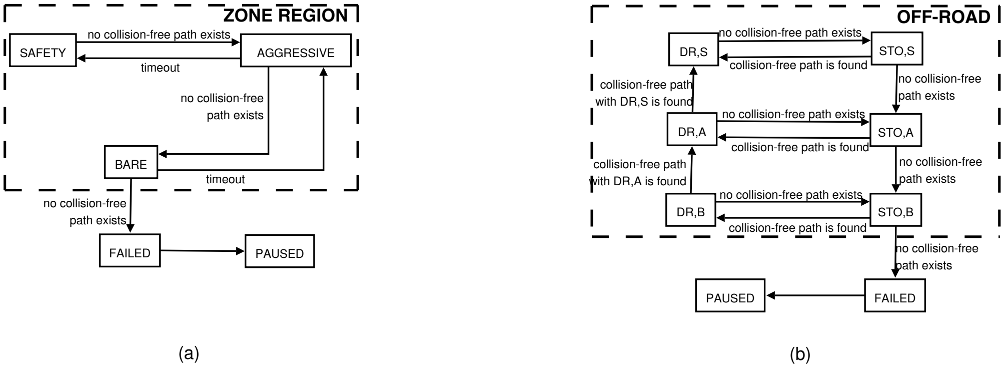
ZONE REGION
SAFETY
no collision-free path exists
AGGRESSIVE
timeout
no collision-free path exists
BARE
timeout
no collision-free path exists
FAILED
PAUSED
(a)
OFF-ROAD
DR,S
no collision-free path exists
STO,S
collision-free path is found
collision-free path with DR,S is found
no collision-free path exists
DR,A
no collision-free path exists
STO,A
collision-free path is found
collision-free path with DR,A is found
no collision-free path exists
DR,B
no collision-free path exists
STO,B
collision-free path is found
no collision-free path exists
PAUSED
FAILED
(b)

Figure 4: The logic planner finite state machine for the zone region (a) and off-road (b). Each state defines the drive state (DR ≡ drive, and STO ≡ stop when Alice is at the right distance from the closest obstacle as specified by the associated minimum allowable distace from obstacles) and the minimum allowable distance from obstacles (S ≡ safety, A ≡ aggressive, and B ≡ bare).

case the actuator or sensor fails.

Gcdrive The gcdrive module 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 exeucted 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 into the role of gcdrive was to implement 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 occuring 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.

5 RESULTS

The 2007 DARPA Urban Challenge's National Qualifying Event was split into three test areas, featuring different challenges. In this section, we present the results from Test Area B which was the most challenging test area from the mission and contingency management standpoint. As shown in Figure 5, Test Area B consisted of approximately 2 miles of driving, including a narrow start chute, a traffic circle, narrow, winding roads, a road with cars on each side that have to be avoided and an unstructured region with an opening in a fence, and navigating and parking at a designated spot in an almost fully occupied parking lot.

5.1 Attempt 1

We started the NQE with a reasonably conservative vehicle separation distance as specified by DARPA. As a result, the logic planner spent a considerable amount of time in the aggressive and bare states as shown in Figure 6. Alice had difficulties finishing this course mainly due to the vehicle separation distance problem which caused Alice to spend about five minutes trying to get out of the start chute area and more than ten minutes trying to park correctly according to the DARPA's specification while keeping the required minimum distance from obstacles. Specifically, the problem was that in the start chute area, there were K-rails less than one meter away from each side of Alice, which was illegal according to the DARPA rules. Alice had to progress through a se-

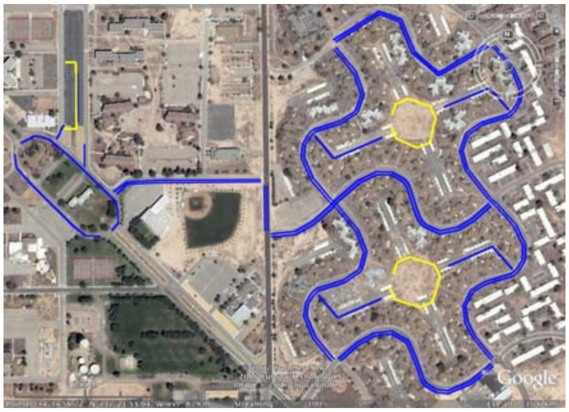

Figure 5: Test Area B

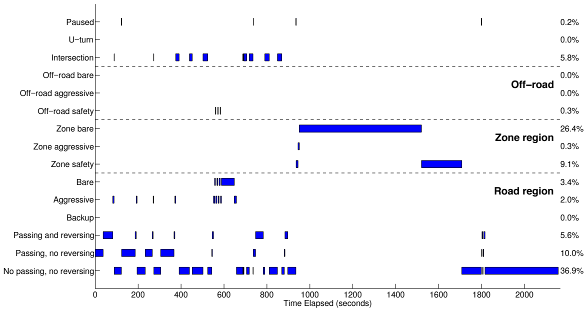

Figure 6: The logic planner state during NQE Test Area B run #1.

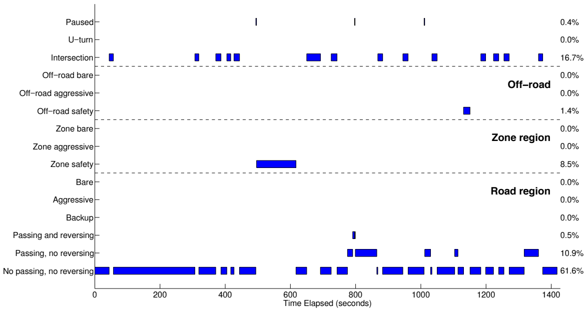

Figure 7: The logic planner state during NQE Test Area B run #2.

ries of internal planning failures before finally driving with reduced buffers on each side of the vehicle. In the parking lot, there was a car parking right in front of our parking spot and if Alice was to park correctly, she would have to be within two meters of that car; thus, violating the required minimum distance from obstacles as specified by DARPA. Alice ran out of the thirty minute time limit shortly after we manually moved her out of the parking lot.

Despite the failure in completing this run within the time limit, Alice demonstrated the desired behavior, consistent with what we have seen in over two hundred miles of extensive testing, that she would keep trying different strategies in order to get closer to completing the mission and she would never stop as long as the system is capable of operating safely. Had she been given enough time, the mission control would have detected the lack of forward progress and decided to skip the parking and continue to complete the rest of the mission.

5.2 Attempt 2

After the first run, we decided to decrease the required vehicle separation distance and relax the tolerance of reaching waypoints so Alice could complete the course faster. Alice was then able to successfully complete the course within twenty three minutes with only minor errors. The logic planner state during the second attempt is shown in Figure 7. As expected, in the second run, the logic planner never transitioned to the aggressive or bare state.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

We describe Team Caltech's approach to mission and contingency management for the 2007 DARPA Urban Challenge. This approach allows mission and contingency management to be accomplished in a distributed and dynamic manner. It comprises two key elements—a mission management subsystem and a Canonical Software Architecture for a planning subsystem. The mission management subsystem works in conjunction with the planning subsystem to dynamically replan in reaction to contingencies. 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. 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. Extensive testing has demonstrated the desired behavior of the system which is that it will keep trying different strategies in order to get closer to completing the mission and never stop as long as the it is capable of operating safely.

6.2 Future Work

As described in Section 2, a controlled module can only report failure, rejection, acceptance and completion of a received directive. The CSA does not incorporate the notion of uncertainty in the response. In other words, a response can only be "I can do it" or "I can't do it" but not "With a probability of 0.5, some constraints may be violated if I execute the directive." This notion of uncertainty is important especially when sensing data is noisy. Consider a scenario where spurious obstacles are seen such that they completely block the road. Although the map may correctly have high uncertainty, the logic planner will still progress through all its states before finally concluding that it cannot complete the segment-level goal. Failure will then be reported to the mission planner which will incorrectly evaluate the current situation as the road is completely blocked and subsequently plan a U-turn. If the response also incorporates a notion of uncertainty, the mission planner can use this information together with the system health and issue a pause directive instead so Alice will stop and wait for better accuracy of the map.

Another direction of research is to formally verify that if implemented correctly, the directive/response mechanism will guarantee that the states of different software modules will be consistent throughout the system and that the CSA and the mission management subsystem guarantee that Alice will keep going as long as it is safe to do so. Using temporal logic, we were able to formally verified the state consistency for the follower and gcdrive modules as previously described in Section 4. For the rest of the system, we have only verified the state consistency and the fail-ops/fail-safe capability through extensive testing.

Lastly, it is also of interest to verify that this distributed mission and contingency management approach actually captures all the functionality of a centralized approach such as SuperCon and that it actually facilitates formal verification of the system. We believe that this is the case for many systems in which the central module does not take into account the uncertainties in the system and the environment.

ACKNOWLEDGEMENTS

The idea of the CSA came from discussions with Robert Rasmussen and Michel Ingham and was implemented by Josh Doubleday from the Jet Propulsion Laboratory. The health monitor module was developed by Chris Schantz. The following individuals have contributed to the development of the planning subsystem: Vanessa Carson, Noel duToit, Sven Gowal, Andrew Howard, Magnus Linderoth, Christian Looman, Kenny Oslund, Kristian Soltesz. Special thanks go to the members of Team Caltech without whose contributions this work would not have been possible.

This work was supported in 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.

REFERENCES

Antonelli, G. (2003). A survey of fault detection/tolerance strategies for auvs and rovs. In Caccavale, F. and Villani, L., editors, *Fault Diagnosis and Fault Tolerance for Mechatronic Systems: Recent Advances*, volume 1 of *Springer Tracts in Advanced Robotics*, pages 109–127. Springer Berlin / Heidelberg.

Barrett, A., Knight, R., Morris, R., and Rasmussen, R. (2004). Mission planning and execution within the mission data system. In *Proceedings of the International Workshop on Planning and Scheduling for Space*.

Braid, D., Broggi, A., and Schmiedel, G. (2006). The Terramax autonomous vehicle. *Journal of Field Robotics*, 23(9):693–708.

Chen, Q. and Ümit Özgüner (2006). Intelligent off-road navigation algorithms and strategies of team desert buckeyes in the darpa grand challenge 2005. *Journal of Field Robotics*, 23(9):729–743.

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

Dearden, R., Hutter, F., Simmons, R., Thrun, S., Verma, V., and Willeke, T. (2004). Real-time fault detection and situational awareness for rovers: Report on the mars technology program task. In *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT.

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

Franke, J., Hughes, A., and Jameson, S. (2006). Holistic contingency management for autonomous unmanned systems. In *Proceedings of the AUVSI's Unmanned Systems North America*, Orlando, FL.

Franke, J., Satterfield, B., Czajkowski, M., and Jameson, S. (2002). Self-awareness for vehicle safety and mission success. In *Unmanned Vehicle System Technology*, Brussels, Belgium.

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

Jones, J. L. and Roth, D. (2004). *Robot Programming: A Practical Guide to Behavior-Based Robotics*, chapter 4. McGraw-Hill.

Linderoth, M., Soltesz, K., and Murray, R. M. (2008). Nonlinear lateral control strategy for nonholonomic vehicles. In *Proceedings of the American Control Conference*. Submitted.

Looman, C. (2007). Handling of dynamic obstacles in autonomous vehicles. Master's thesis, Universität Stuttgart.

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

Rasmussen, R. D. and Ingham, M. D. personal communication.

Williams, B. C., Ingham, M. D., Chung, S. H., and Elliott, P. H. (2003). Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, volume 9, pages 212–237.