CajunBot Path Planner Architecture for Autonomous Ground Vehicles in an Urban Environment

A Dissertation

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Suresh Golconda

Spring 2010

CajunBot Path Planner Architecture for Autonomous Ground Vehicles in an Urban Environment

Suresh Golconda

APPROVED:

Arun Lakhotia, Co-Chair Professor of Computer Science The Center for Advanced Computer Studies	Anthony S. Maida, Co-Chair Associate Professor of Computer Science The Center for Advanced Computer Studies
Kemal Efe Associate Professor of Computer Science The Center for Advanced Computer Studies	Rasiah Loganantharaj Associate Professor of Computer Science The Center for Advanced Computer Studies
C. E. Palmer Dean of the Graduate School	

Dedication

To my mom and dad.

Acknowledgments

I am deeply in debt to my advisors, Dr. Arun Lakhotia and Dr. Anthony Maida for their valuable support, patience and guidance throughout my research and dissertation work. Dr. Arun Lakhotia has always been there supporting me and motivating me in my research and pushing me to new limits. I also want to thank him for leading the Team CajunBot and providing all the support and resources that I ever required. This support has given me strength to concentrate on my actual research work.

I thank Dr. Anthony Maida for being one of the greatest mentors in my life. He has been guiding me with my work since my initial days in this country. His patient listening and calm support has always given me the strength to challenge new research problems. I thank him for all his guidance.

I thank Pablo Mejia for his continuous support and for the countless hours he spent discussing different algorithms with me. He has always inspired me with his intelligence and hard work, and most importantly his down to earth attitude despite all his skills. I always try to follow his example and hope to be like him. I consider myself honored to be able to work with him and learn from him.

I can very confidently say that since my days in this country, Dr. Arun Lakhotia, Dr. Anthony Maida, and Pablo Mejia were the three important people who have shaped me both as a researcher and as an individual.

I thank Adrian Aucoin, Jr. and Joshua Bridevaux for the constant support they provided since the very first day in the team, and also for sharing all the tough and fun moments in the team. I thank Josephine McKelvy for the long hours she spent reviewing my write-up and helping me maintain the quality of the dissertation.

I thank Mark McKelvy, Christopher Mire, Dallas Griffith, and John Herpin for their invaluable feedback and their patience in field testing my algorithms on the CajunBot-II.

I thank one and all of the CajunBot Team members and the University of Louisiana at Lafayette for their support for making the CajunBot a success. My present research work would not have been possible without the success of Team CajunBot.

Finally, but most of all, I want to thank my family for the constant support and encouragement they have shown at every step in my life.

Table of Contents

List of Tables

List of Figures

1 Introduction

Autonomous ground vehicles (AGVs) are a promising innovation for saving lives in military and civilian applications. In military applications, these vehicles may navigate through areas that are unsafe for soldiers and carry food and medicines. In civilian applications, AGVs may augment a driver's ability and prevent human error.

The research and development of AGV technology has gained great interest in the last decade, leading to new research interest in both the hardware, and software components of the system. Hardware research includes developing longer range and more accurate environmental analysis sensors, more accurate GPS positioning systems, faster control hardware and better vehicle platform. Software research includes developing more efficient and innovative approaches for analyzing the sensor data for better decision-making and gaining better control over the vehicle's navigation system.

Figure 1 shows the dataflow diagram of an AGV's software system for autonomous navigation. A driver provides the software interface to a hardware system. It captures and decodes the input sensor data or manipulates the control hardware, such as an actuator. An obstacle detector module analyzes the sensor data and detects objects or the terrain surrounding the AGV. The World_State module merges results from multiple obstacle detector modules and creates a map of the AGV's environment. The Path Planner module analyzes the situation and plans a path to direct the AGV towards its mission. The Steering Controller module generates the low-level steering control commands to navigate the vehicle along the Path Planner's path.

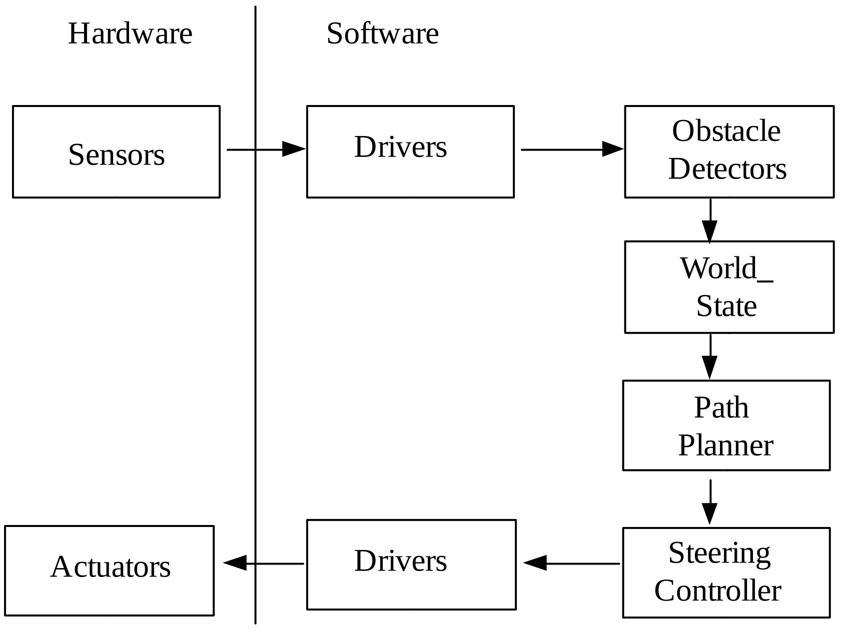


Figure 1: Data flow diagram of a typical AGV's software system

The present dissertation work concentrates on the path planner module of an AGV's software system.

1.1 Motivation

One of the challenges in building an AGV is to give it the ability to plan its path and make navigation decisions when required. The software module named *Path Planner* provides these capabilities to an AGV. Its purpose is to navigate an AGV safely through an urban environment while obeying a set of traffic rules. Figure 2 shows the black box representation of the Path Planner module. It takes as input: the route description, the mission, the AGV's instantaneous position and orientation, and the environment information. Here, a route description is a list of navigable regions of an urban environment. A mission is a list of checkpoints that the AGV should drive through and the environment information includes the status of other vehicles and lane blockages. The Path Planner accesses these inputs from the

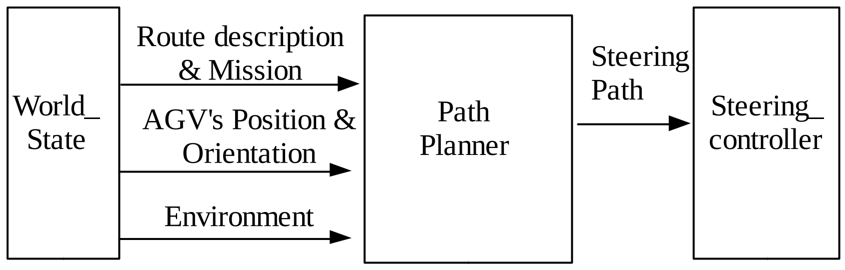


Figure 2: Black box diagram of the Path Planner

World_State module which maintains all the information about the environment, such as a list of static blockages on a lane and a list of vehicles stopped at an intersection. The Path Planner outputs a steering path that is a sequence of equally spaced GPS points that the AGV should follow. Each point in the steering path is labeled with a maximum speed to drive through that point. The Path Planner takes in the latest input data and publishes a new steering path at regular intervals. Each of these iterations is called a *path-planning cycle*.

The scenarios that the Path Planner needs to handle for urban driving are open-ended, especially due to the uncertainty in the other vehicles' behaviors, weather conditions, and sensor failures. Developing a Path Planner to handle all possible scenarios is extremely difficult. More so, the required capabilities may vary from state to state, based on the local state traffic rules, and the Path Planner should support easy addition or deletion of these capabilities to support changing requirement.

The inspiration to build such a system originated from the Hollywood movie 'The Matrix'. In this sci-fi movie, humans are plugged into a computer simulated world named the 'Matrix' where they live. In the simulated world, all of their capabilities are provided through computer programs. In one of the scenes, the main character who had to fly a helicopter loads a flying program into her system. This immediately gives her the capability to fly a helicopter

Figure 3: CajunBot-II

while not affecting any of the existing capabilities. This has been an inspiration to develop a Path Planner architecture that allows easy plugging in of new capabilities without affecting the existing capabilities. As an example scenario, a stable version of Path Planner that can navigate an AGV through an urban environment should be able to easily plug-in the capability to parallel park. The new plug-in should integrate without major modifications and have a minimal effect on the existing system.

The present planner provided the path planning and decision making capabilities to CajunBot-II (Figure 3) (CajunBot Lab 2009), an AGV developed at the CajunBot Lab of the University of Louisiana at Lafayette, to participate in the DARPA's Urban Challenge (UC) ((DARPA) 2007). In the rest of this document, I will refer to the planner as *CB_PP* (CajunBot Path Planner).

1.2 Problem Statement

This dissertation designs a Path Planner architecture to navigate an AGV through an urban environment, while providing easy plug-ability of new capabilities with minimal or no affect to the existing capabilities. The planner should be able to navigate the AGV at speeds up to 40kph and be able to Re-plan and continue the mission if required.

1.3 Contribution

I propose a Path Planner architecture that allows easy addition of new capabilities while minimally affecting the existing capabilities. In this process I propose a Base-Path protocol, that allows multiple planners planning their paths for an AGV to be completely oblivious of other such planner's implementation and still contribute to one final path that an AGV can follow.

I introduce a novel approach for checking for the safety against dynamic obstacles while following a path in the urban environment. This checking is usually time consuming as it typically involves iterating in small time steps to estimate the position of the AGV and the other dynamic obstacles in future and checking for the safety. I introduce a different approach for checking for safety against dynamic obstacles. Utilizing the urban road network, and assuming that the dynamic obstacles (vehicles) follow traffic rules, it is realized that there are only a few spots through which a dynamic obstacle can enter and interfere with the AGV's path. This observation is used to introduce a new approach for checking safety against dynamic obstacle.

I introduce a novel path planning approach for zones. The planner tries to capture the benefits of two different classical path planning approaches, namely, the grid-based approach (Barraquand, Langlois and Latombe 1991) and Dubin's based approach (Dubins 1957) (Agarwal, Prabhakar and Hisao 1995). The grid-based approaches are typically less time consuming but the extracted paths are not smooth because of rounding of planning resolution to cells. A Dubin's based continuous space exploration generates smooth paths but is computationally expensive. The new path planning approach introduced for zone merges both the approaches and plans a smooth path efficiently.

1.4 Organization

Section 1 introduces some background concepts that would help in understanding the rest of the dissertation. It introduces the STRIPS & ADL concept that is later used in Section 6.3.1. It then Introduces the Dubin's car model-based reachability tree exploration concept which is used in Section 4.5. This exploration concept is used to search a path for driving in open-area zone. Finally, it introduces to the terminology that would be used frequently in the dissertation.

Section 1 gives an overview of the Path Planner architecture and introduces its components. The introduced Path Planner is then illustrated to achieve an example mission. The Path Planner module has one point stop for all the information it requires about the elements in its environment. This is provided by the World_State module which is introduced in this section.

Section 4 introduces basic capabilities and provides some guidelines on how to draw a line between basic capabilities definition. It introduces Basic Planners (BPs), which provide these basic capabilities, as well as Base-Path protocol, which allows transparency between the BPs. This transparency helps in achieving easy plug-ability of BPs without affecting the existing ones. The section then provides an overview on one of the BP, Zone-Navigator BP, because of its novel approach of combining the grid-based planning and Dubin's based continuous space path exploration approach.

Section 5 introduces behavioral capabilities, which are provided by the Behavioral Planner (BhP). The section shows how a collection of BhPs, which form the Re-planner, is represented as a state-machine. It provides how the state-machine representation allows easy plug-ability of BhPs. The section also provides some possible updates to use BhPs to handle dynamic obstacles on the lanes and to handle situations encountered within a zone.

Section 6 introduces the High-Level Planner (HLP) which determines the sequence (σ) of BPs that can achieve a given mission. It proposes how plug-ability of BPs can be achieved in HLP using logic-based language representation. It represents a sample problem statement in ADL logic language and shows how a forward state-space search algorithm can be used to plan a BP sequence (σ) for a mission. It then introduces decision-rule based approach for planning a BP sequence that was implemented and tested for the Urban Challenge.

Section 7 introduces the Supervisor module which uses the BP sequence (σ) planned by HLP to plan a steering path. It provides a description of how the Supervisor module maintains the BP sequence (σ) updated, triggers the BPs to plan their path; checks for the safety of the planned path against static and dynamic obstacles, and finally how it collects the paths planned by all the BPs to publish as a single steering path.

Section 8 evaluates the present system by providing some notes on its testing and its performance at the DARPA’s Urban Challenge (UC). The section then compares the present approach with that of the winner of UC, Boss from Carnegie Mellon University, second prize winner, Junior from Stanford University, and Skynet from Cornell University.

2 Background

This section introduces some background concepts that would be helpful in the rest of the dissertation. The section first introduces the STRIP & ADL language for problem representation. This is referred to from Section 6.3.1. The section then introduces the Dubin's car model based reachability tree exploration, used in Section 4.5. And finally, introduces the terminology that is used in the rest of the dissertation.

2.1 STRIPS & ADL

This section gives a brief introduction to STRIPS followed by extending it to ADL language. ADL language represented is used to explain the possible approach for the BP Extractor module of High-Level planner in Section 6.3.1.

STRIPS is a language for representing logic based planning problems. It stands for Stanford Research Institute Problem Solver, developed in 1971 by Nils J. Nilsson and Richard E. Fikes. The language representation has three parts, namely: representations of states, representation of a goal, and representation of actions.

States are represented as a conjunction of positive literals such as $\mathrm{Sick} \wedge \mathrm{Sleeping}$ or using first-order literals such as, $\mathrm{At}\ (\mathrm{John}, \mathrm{Saint\_Hospital}) \wedge \mathrm{Sleeping}\ (\mathrm{John})$. STRIPS does not allow nesting of first-order literals, such as $\mathrm{At}\ (\mathrm{Sleeping}\ (\mathrm{John}), \mathrm{Saint\_Hospital})$.

A Goal is represented as a conjunction of positive literals or first-order literals which should be true. An example goal state includes: Discharge (John, Hospital).

The action representation has three parts: signature, pre-conditions and effects. Here, the signature specifies the name of the action and the arguments it takes, the pre-conditions specify the conditions that should be true for the action to take place, and the effects specify the changes to the state. Following are two example actions:

Action (*Admit* (x, h),
    PRECOND: $\neg At(x, h) \wedge Person(x) \wedge Hospital(h) \wedge Sick(x)$
    EFFECT: $At(x, h)$

Action (*Discharge* (x, h),
    PRECOND: $Person(x) \wedge Hospital(h) \wedge At(x, h) \wedge \neg Sick(x)$
    EFFECT: $\neg At(x, h)$

In the STRIPS language, variables are represented using lowercase letters while, constants, literals, first-order literals, and action names start with capital letters.

The STRIPS formalism gained a lot of popularity and led to the development of many new languages. One such new language which gained importance is Action Description Language (ADL). ADL has more expressive power compared to STRIPS. To mention a few modifications to ADL from STRIPS, an ADL allows negative literals in state representation, such as ¬John, ¬Saint_Hospital. ADL allows quantified variables while representing its goals, such as $\exists x$ Person (x) ∧ ¬Sick (x) represents a goal of no one is sick. ADL supports equality (h_1=h_2) predicates. A more detailed list of ADL extension can be found in(Russell and Norvig, The Planning Problem 2003).

2.2 Dubin's Car Model Based Reachability Tree

This section describes usage of Dubin's car model to explore the reachability tree of a car (La Valle 2006). This reachability tree is used in Section 4.5 to explore and search for a path for driving in a zone.

Here, the state s, of the car is represented by its position and orientation, $s = (x, y, \theta)$. Assuming a car can perform either a fixed left-turn, straight-drive or a fixed right-turn maneuver from any state, a given state s can lead to three new states, namely s_L, s_S, and s_R. Here, s_L represents the state reached by performing a left-turn by an angle $\Delta\theta$, and, s_R represents the state reached by performing a right-turn by an angle $\Delta\theta$. Figure 4 (a) figuratively shows these states. Assuming a constant speed *sp*, and exploring the steps for regular time interval Δt, the distance covered by the car is given as $d = sp * \Delta t$. The parameters of the new states reached can be calculated using following equations:

$$s_L.x = s.x + d * \cos(s.\theta + \Delta\theta)$$

$$s_L.y = s.y + d * \sin(s.\theta + \Delta\theta)$$

$$s_L.\theta = s.\theta + \Delta\theta$$

$$s_S.x = s.x + d * \cos(s.\theta)$$

$$s_S.y = s.y + d * \sin(s.\theta)$$

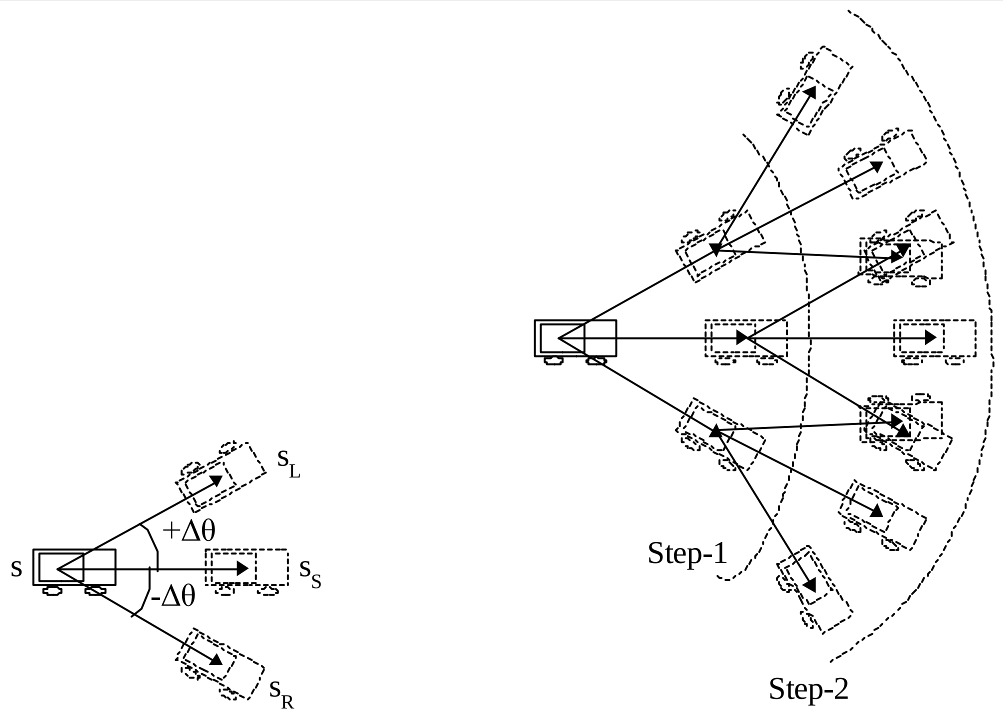
$$s_S.\theta = s.\theta$$

$$s_R.x = s.x + d * \cos(s.\theta - \Delta\theta)$$

$$s_R.y = s.y + d * \sin(s.\theta - \Delta\theta)$$

$$s_R.\theta = s.\theta - \Delta\theta$$

Figure 4 (b) shows the reachability tree exploration for two steps. With three possible actions from each state, a state explores out to 3 states, a two step exploration leads to reachability tree with 3 * 3 = 9 states. In general an n step exploration leads to 3^n states. The parameter $\Delta\theta$ is set to match the driving capability of the car, and Δt is set to specify how detailed path exploration is to be performed. A low Δt value helps to achieve a more detailed path exploration but at a cost of increased processing time. A high Δt value achieves a less detailed path exploration at reduced processing time. One way to compromise between detail path and processing time is to have a varying Δt. One possible approach could be to have a smaller Δt for initially exploration steps, and larger Δt for later exploration steps. The initial shorter Δt helps to plan a more detail immediate path, and longer Δt plans a less detail path for future.



(a) Exploring states s_L, s_S and s_R from state *s*. (b) Two steps of exploration.

Figure 4: Dubin's based path exploration steps

While exploring a reachability tree, a state can be reached more than once, suggesting that a state can be reached via more than one path. Such duplicate states can be handled by discarding the copy of the state with longer path. But this comparison for duplicate states itself might be costly with the increased number of states, and hence duplicate states are ignored.

2.3 Terminology

This section introduces some of the terms frequently used in rest of the document. Some of these terms were introduced as part of the technical specifications of DARPA’s Urban Challenge (DARPA's Urban Challenge 2009).

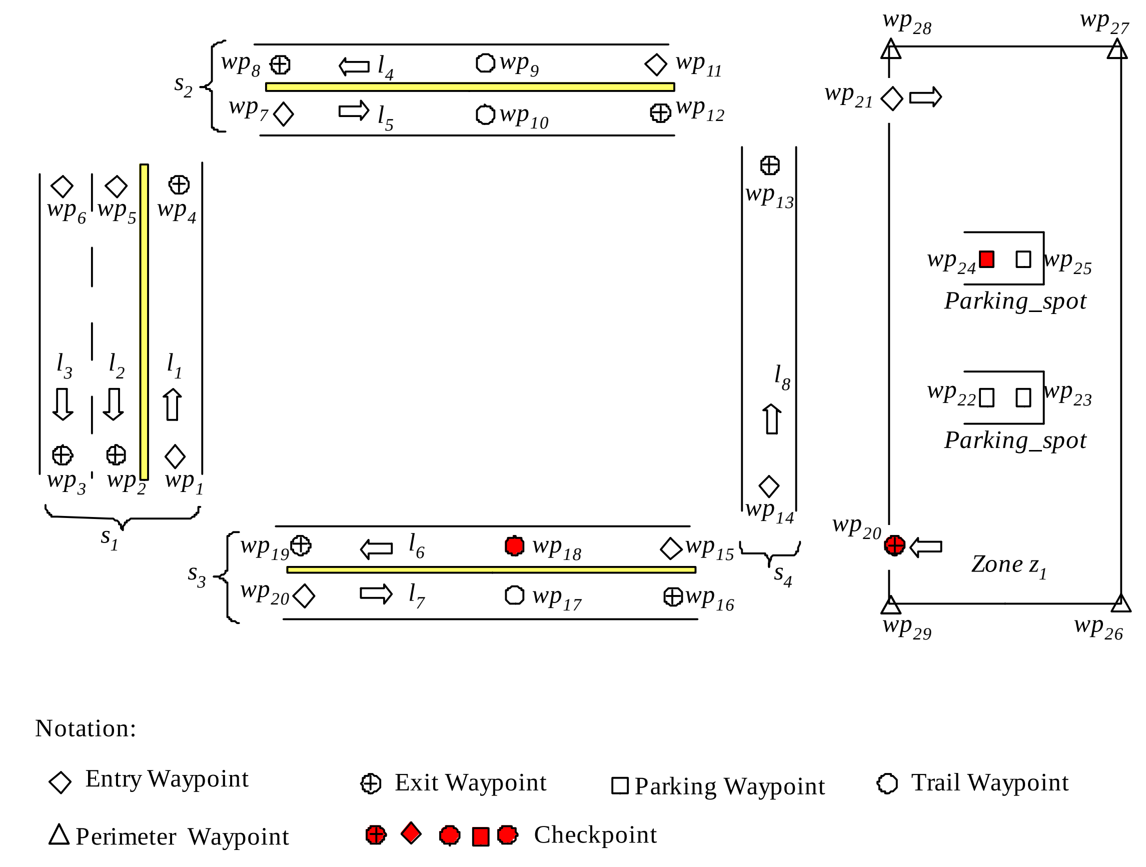


Figure 5: An example urban environment with RNDF notation

Here, I represent constant symbols in capital letters, such as *L*, *S*, and *WP* and variable symbols which will take a constant value, in small letters, such as *l*, *s*, and *wp*.

RNDF: Stands for Route Network Definition File. It describes the navigable regions of an urban environment. It contains two regions, namely *Segments* and *Zones*. A Segment *s* is a collection of lanes *l* that represents the navigable regions on a road. A zone *z* is a free-travel area such as a parking-lot with possible parking-spots. These regions are defined using *Waypoints wp*, which specify a location in a physical space. The location of a waypoint is normally specified in the GPS coordinate system. Figure 5 shows an example urban environment with RNDF labeling notation.

Waypoints are used to represent the position and shape of the segments and zones. For example a lane (*l*) is defined using a sequence of waypoints, l = [wp_1, wp_2,…wp_n]. Similarly a sequence of waypoints known as perimeter waypoints $[pr\_wp_1, pr\_wp_2, \dots\ pr\_wp_n]$ is used to define the boundary of a zone. The position and orientation of an allowed parking area in a zone is defined by a pair of waypoints called parking-spot, *pk_wp*[2]. A zone (*z*) is defined by the sequence of perimeter waypoints and a list of parking-spots it contains, as expressed below.

$$z = \left(\left[pr, pr, \dots\ pr_{wp_n}\right],\ \{pk\_wp_1[2], pk\_wp_2[2] \dots\ pk\_wp_n[2]\} \right)$$

A lane (*l*) is attributed with parameters such as {*direction*, *width*, *left boundary marking*, and *right boundary marking*}. Here a 'direction' can either be set to *primary* or *secondary*. This '*direction*' parameter is used to represent the set of lanes that are going in the opposite direction. '*Width*' provides a rough lateral distance between its left and right boundaries. '*Left boundary marking*' and '*right boundary marking*' can take one of the follow values: *double_yellow, solid_yellow, solid_white*, *broken_white*, and *unknown*. These values represent if a lane change is allowed between neighboring lanes.

A waypoint (*wp*) is attributed by five non-exclusive flags namely: *ex, en, cp, pr,* and *pk.* In the rest of the document the *wp*s with these flags set are represented by symbols *ex_wp*, *en_wp*, *cp_wp*, *pr_wp*, and *pk_wp* respectively. *EX_WP*, *EN_WP*, *CP_WP*, *PR_WP*, *PK_WP* represent the set of respective waypoints. Below is the mathematical representation of the sets. Here WP is the set of all the waypoints in the RNDF description.

$$EX\_WP = \{wp \mid wp \in WP \wedge wp.ex = true\}$$

$EN\_WP = \{wp \mid wp \ \in WP \ \wedge wp.en = true\}$

$CP\_WP = \{wp \mid wp \ \in WP \ \wedge wp.cp = true\}$

$PK\_WP = \{wp \mid wp \ \in WP \ \wedge wp.pk = true\}$

$PR\_WP = \{wp \mid wp \ \in WP \ \wedge wp.pr = true\}$

A set of waypoints included on the lane just to provide the shape of the lane are called *Trail waypoints*. These waypoints can be specified as a checkpoint. *Trail waypoints* = *WP* – *EX_WP* – *EN_WP* – *PK_WP* – *PR_WP*. For the example urban environment shown in Figure 5 contents of these sets is listed below.

WP = $\{wp_1, wp_2, \ldots wp_{29}\}$

$EX\_WP = \{wp_2, wp_3, wp_4, wp_8, wp_{12}, wp_{16}, wp_{19}, wp_{20}\}$

$EN\_WP = \{wp_1, wp_5, wp_6, wp_7, wp_{11}, wp_{14}, wp_{15}, wp_{20}\}$

$CP\_WP = \{wp_{18}, wp_{20}, wp_{25}\}$

$PK\_WP = \{(wp_{22}, wp_{23}), (wp_{24}, wp_{25})\}$

$PR\_WP = \{wp_{26}, wp_{27}, wp_{28}, wp_{29}\}$

Trail waypoints = $\{ wp_9, wp_{10}, wp_{17}, wp_{18}\}$

An *ex_wp* denotes a *wp* from which a lane or a zone can be exited to enter another zone or a lane through its *en_wp*. A en_*wp* can also be attributed with stop sign where the vehicles are stopped to reach complete stop before proceeding. The function *stop_sign* (*wp*) is true if the *wp* has stop sign, else is false. An *ex_wp* stores a list of these *en_wp*s to which it is connected to. Similarly an *en_wp* contains a list of *ex_wps* from which it can enter the region it belongs to. A *cp_wp* denotes a *wp* that can be included in the input mission as one of the goal positions to drive to. And finally, as described earlier, a perimeter point (*pr_wp*)

Table 1: RNDF description of the urban environment from Figure 5

Segment	Lane l	*Waypoint(l)*	*Direction*	*Width (m)*	*LBM*	*RBM*
s_1	l_1	$[wp_1, wp_4]$	*primary*	2	*solid_yellow*	*solid_white*
	l_2	$[wp_5, wp_2]$	*Secondary*	2	*solid_yellow*	*broken_white*
	l_3	$[wp_6, wp_3]$	*secondary*	2	*broken_white*	*solid_white*
s_2	l_4	$[wp_{11}, wp_9, wp_8]$	*primary*	2	*solid_yellow*	*solid_white*
	l_5	$[wp_7, wp_{10}, wp_{12}]$	*secondary*	2	*solid_yellow*	*solid_white*
s_3	l_6	$[wp_{15}, wp_{18}, wp_{19}]$	*primary*	2	*solid_yellow*	*solid_white*
	l_7	$[wp_{20}, wp_{17}, wp_{16}]$	*secondary*	2	*solid_yellow*	*solid_white*
s_4	l_5	$[wp_{14}, wp_{13}]$	*secondary*	2	*solid_white*	*solid_white*

Zones

Zone	Perimeter Waypoints	Parking spots
z_1	$[wp_{26}, wp_{27}, wp_{28}, wp_{29}]$	$[(wp_{22}, wp_{23}), (wp_{24}, wp_{25})]$

LBM = Left boundary marking, *RBM = Right boundary marking*

specifies the boundary of a zone and a parking spot, *pk_wp* specifies an allowed parking area in the zone. A parking-spot is always specified in a pair, *pk_wp*[2] representing the position and orientation of the parking area.

An *ex_wp* is attributed by two parameters, if it has a stop sign and what are the of entry waypoints that are navigable from *ex_wp*. The function *stop_sign* (*ex_wp*) returns return true if the *ex_wp* has a stop sign, else returns false. The function entry_pair (*ex_wp*) returns the list of *en_wp* that are connected to *ex_wp*. This exit-entry pair represents the allowed navigable areas in an urban environment.

A given waypoint *wp* can be denoted by more than one symbol, that is it could be both an exit waypoint (*ex_wp*) and a checkpoint (*cp_wp*).With the above introduces terminology I can formally define a rndf as below. Table 1 describes the example urban environment from Figure 5 using this notation.

RNDF = (S, Z)

$S = \{s_1, s_2 .. s_n\}$

$Z = \{z_1, z_2, \ldots z_m\}$

$s_i = [l_1, l_2.. l_k]$

$Waypoints(l_i) = [wp_1, wp_2 \ldots wp_q]$

$z_i = ([pr\_wp_1, pr\_wp_2, \ldots\ pr\_wp_n],\ \{pk\_wp_1[2], pk\_wp_2[2] \ldots\ pk\_wp_m[2]\})$

MDF: Stands for Mission Data File. It describes the mission that an AGV needs to accomplish. The MDF mission is defined by a sequence of checkpoints $[cp_1, cp_2, \ldots cp_n]$ that an AGV should reach in the listed order. The MDF also describes the speed limits that the AGV should follow on each segment s and zone z. An example MDF mission for an urban environment described in Figure 5 is the sequence $[wp_{18}, wp_{24}]$.

Path (*P*): Is a sequence of equally spaced GPS points (*p*) that an AGV tries to follow. A point (*p*) is annotated with the GPS position (*x*, *y*) and speed limit *max_speed*. A path *P* is attributed by *direction* of the path. The direction *d* can take values 'forward' or 'reverse', specifying if the path should be followed in forward gear or reverse gear of the vehicle. In the present notation capital 'P' represents the path and small 'p' represents a point on the path. With this terminology path *P* can be defined as:

$P = ([p_1, p_2, \ldots p_n], directon)$

$p = (x, y, max\_speed)$

$direction \in \{forward, reverse\}$

3 Path Planner Architecture

This section gives an overview of the Path Planner architecture and introduces its components. These components of the Path Planner are then explained in more detail in the following sections. Section 3.2 illustrates how an example mission with two checkpoints can be achieved by the present Path Planner. Section 3.3 introduces the World_State module that provides a one point interface to all the information that the Path Planner would need.

3.1 Overview of the Path Planner Architecture

Since the scenarios that an AGV needs to handle for urban driving are open-ended, designing a Path Planner is extremely difficult. Hence, the present planner (CB_PP) is built to address a set of well defined scenarios specific to DARPA’s Urban Challenge (UC), while also maintaining an architecture that allows easy addition of new capabilities without affecting the existing capabilities. The UC’s technical document lists the capabilities required for the challenge (Urban Challenge: Technical specifications 2009). Some of these capabilities include: planning the quickest path to reach a given sequence of checkpoints by safely following a lane, switching lanes when required, negotiating with traffic vehicles at intersections, driving in free-travel areas (zones) while avoiding obstacles to reach the parking spots, parking, handling lane blockages, and re-planning the mission when required.

The capabilities required of CB_PP are categorized into two levels: *basic-capabilities* and *behavioral-capabilities*. A basic-capability includes the ability to perform tasks such as following a lane or changing to a neighboring lane. Each basic-capability is achieved by a specialized planner called a *Basic-Planner* (BP). Similarly a behavioral-capability includes the ability to use BPs to handle the unexpected situations encountered while following the path. Each behavioral-capability is achieved by a planner called a *Behavioral-Planner* (BhP). The collection of all the BhPs is called the Re-Planner. Along with the Basic-Planners and the Re-Planner module, a High-Level Planner (HLP), and a Supervisor module constitute the CB_PP. Each of these modules are introduced in the rest of the section.

A BP provides its capability by planning a path (p) for the AGV to follow. While each BP provides one basic-capability, a sequence σ, of these BPs, determined at the beginning of a mission, plan the initial path for accomplishing the complete mission. During execution of the mission, this path is modified as unanticipated situations are encountered.

A behavioral-capability, on the other hand, involves utilizing the BPs to handle the situations encountered while following the path. For example, if the AGV finds that the lane it is following is blocked ahead, the AGV can use a neighboring lane to get around the blockage or use an oncoming traffic lane. Being able to perform one such action is called a behavioral-capability. Each behavioral-capability is implemented by a separate planner, called a Behavioral-Planner (BhP). Each BhP implements its behavioral-capability by modifying the BP sequence (σ) that is planning the path. This may include adding/removing BPs from the sequence or changing the goal of a BP or changing the speed limits of the already planned path. The BhP can use HLP module to generate this modification to BP sequence.

The initiation and transition between BhPs is represented as a state-machine (Figure 19). A state here represents either the CB_PP's belief of what situation the AGV is in, or represents the behavioral capability being exhibited using a BhP. An example belief state includes safe state and confirm_obstacles state and an example behavioral capability state includes ‘use_neighboring_lane’ state and ‘use_traffic_lane’ state.

The transition edges represent the conditions for exhibiting a behavioral capability or for changing the AGV’s belief about situation. The BhPs together with the state-machine is called the *Re-Planner* module.

CB_PP architecture permits easy addition of basic capabilities and behavioral-capabilities incrementally. Easy addition of new basic-capabilities without affecting the performance of the existing ones is achieved by providing:

a) *Common Interface*: All BPs are extensions of a common interface template (base-class hierarchy).

b) *Confined operation*: Within the template interface, the BPs are confined to plan a path for its own capability.

With each BP having a *common interface* and *confined operation,* it is necessary to ensure that a sequence of BPs plan one combined path that an AGV can follow. This requires that each BP in the sequence should know the ‘state of the vehicle’ when the AGV is going to start following its path. Here a ‘state of the vehicle’ includes, position, orientation and rotation of the vehicle. This is achieved in CB_PP architecture through the ‘*Base-Path protocol*’, which defines the protocol for communication between the BPs.

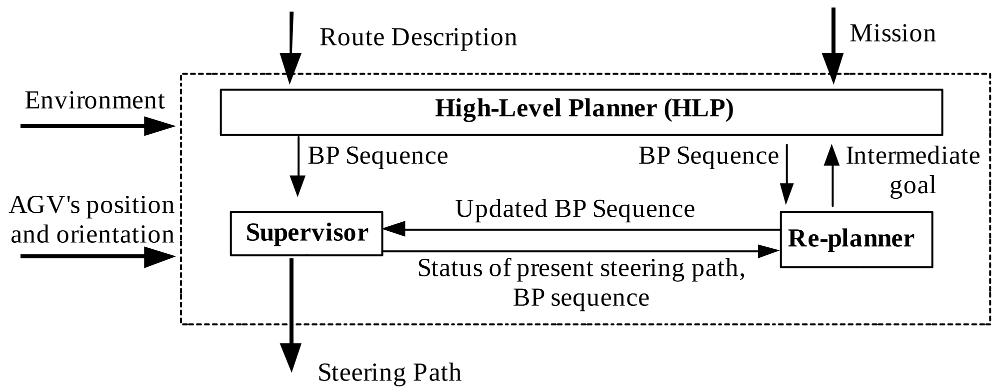


Figure 6: Interaction of Re-Planner, High-Level Planner and Supervisor modules

The Base-Path protocol requires that each BP provides its capability by accepting an input path and extending it to a new path that an AGV can follow. The extended path from one BP is then used as an input path for the next BP in the BP sequence (σ). Thus, even by keeping each BP unaware of other BPs, the sequence of BPs can plane a path that is drivable. Having the BPs unaware of other BPs forms the key element for achieving an easy addition of new basic-capabilities without affecting the existing ones.

For easy addition of new behavioral-capabilities without affecting the existing ones, each BhP providing a behavioral-capability is represented as a separate state in the state-machine. In this representation, adding a new behavioral-capability involves adding a new state representing this capability to the state-machine. A state is added to the state machine by connecting it to the relevant states via edges representing the conditions when the capability can be utilized and what state to return to after implementing the capability. Adding a new state to the state-machine does not affect the state transitions of the existing states, except for the case when the conditions for transitioning to the newly added state satisfy along with the conditions for transitioning to an already existing state. Such situations can be handled by prioritizing such state transitions.

A High-Level Planner, at the beginning of a mission determines an initial sequence of (σ) of BPs that can accomplish the given mission on given route description. This sequence of BPs can be later modified by the Re-Planner to exhibit a behavioral capability. The Supervisor module uses this sequence of BPs at every 50 ms (path-planning cycle) to plan a steering path for the AGV to follow. The Supervisor module also checks for the safety of the planned path against static and dynamic obstacles. Detection of a static obstacle along the path is handled by informing the Re-planner module, which handles the situation using one of its behavioral capabilities. Detection of dynamic obstacles along the path is handled by modifying the speed limits along the path to exhibit behaviors such as ‘convoy a vehicle’ or ‘stop behind a stalled vehicle’.

3.2 Simple Planning Scenario

This section illustrates how a sequence (σ) of individual Basic Planners (BPs) achieves a sample mission of driving through two checkpoints. The mission, as shown in Figure 7, is to drive through checkpoint cp_1 followed by checkpoint cp_2. The whole mission is achieved by a sequence of five BPs, namely [fl_bp_1, cl_bp_2, fl_bp_3, it_bp_4, fl_bp_5], planning paths for their localized goal. In the present document a particular instance of a BP is represented in small letters, such as *fl_bp* and the definition class is represented in capital letters, such as FL_BP.

The five BPs used belong to three types of BPs with different goals. The BP types used here are: FL_BP (Follow-Lane Basic Planner) for following a lane, CL_BP (Change-Lane

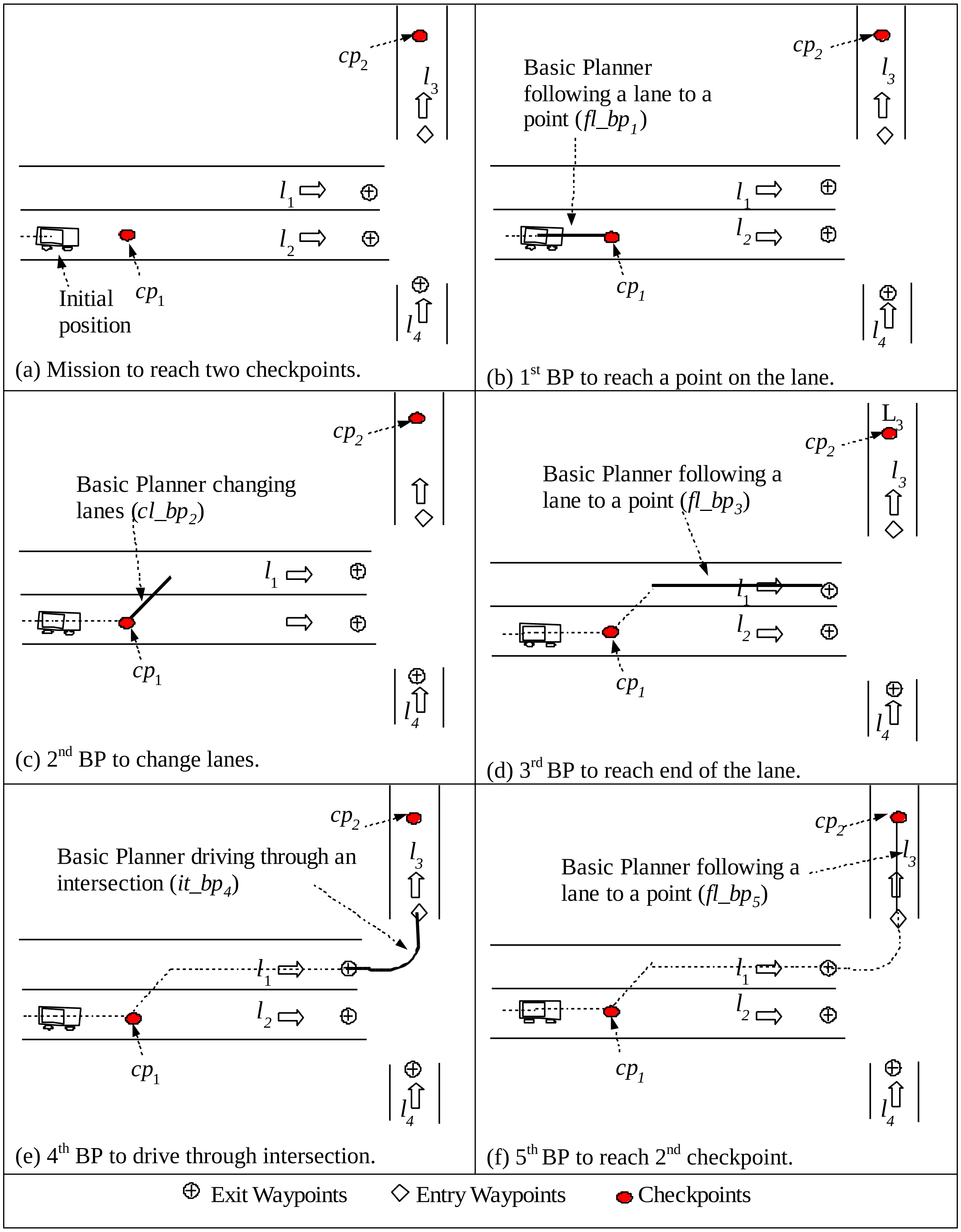

cp_2
l_3
l_1
l_2
Initial position
cp_1
l_4
(a) Mission to reach two checkpoints.
Basic Planner following a lane to a point (fl_bp_1)
(b) 1st BP to reach a point on the lane.
Basic Planner changing lanes (cl_bp_2)
(c) 2nd BP to change lanes.
Basic Planner following a lane to a point (fl_bp_3)
L_3
(d) 3rd BP to reach end of the lane.
Basic Planner driving through an intersection (it_bp_4)
(e) 4th BP to drive through intersection.
Basic Planner following a lane to a point (fl_bp_5)
(f) 5th BP to reach 2nd checkpoint.
⊕ Exit Waypoints
◇ Entry Waypoints
● Checkpoints

Figure 7: Illustration of a sequence (σ) of five BPs achieving a simple mission with two checkpoints

Basic Planner) for changing lanes, and IT_BP (Intersection Basic Planner) for driving through an intersection. Later in the document, I will introduce the complete list of eight BP types that are required for accomplishing the requirements of DARPA's 2007 Urban Challenge.

The mission to drive to cp_1 and followed by cp_2 is shown in Figure 7 (a). This mission requires, continuing on the lane l_2 to reach cp_1, this is achieved by fl_bp_1 as shown in Figure 7 (b). This is followed by switching to lane l_1 which has a valid turn to the future lane of interest l_3. This switching of lane is achieved by cl_bp_2, as shown in Figure 7 (c). Then, continuing on lane l_1 to reach the intersection using fl_bp_3, as shown in Figure 7 (d). Driving through the intersection to enter lane l_3 using it_bp_4 is shown in Figure 7 (e). Finally, continuing on lane l_3 to reach cp_2 using fl_bp_5, as shown in Figure 7 (f).

3.3 World_State

This section explains the World_State module which provides a query based interface of the world elements to the path planner. The World_State provides information such as: shape of lanes, locations and velocities of the obstacles on a lane, and list of stopped vehicles at an intersection. It also provides the RNDF description and the MDF mission which in the rest of the dissertation will be assumed to be already available with the path planner.

The World_State module gathers information from multiple sources and merges data to provide one place with consolidated information about the AGV and the environment surrounding the AGV. The World_State gathers this information once at the beginning of the

Path Planner's path planning cycle. This is performed to avoid giving different results for multiple queries from different the path planner modules.

Table 2 lists some of the interface functions provided by the World_State module. These functions are later used in the dissertation to describe the working of Path Planner's modules. The table lists the function names in the first column and description of the functions in second column.

Table 2: World_State interface functions

Interface function	*Description*
obstacle_detected_for_time (*obstacle_id, timeout*)	Returns true if the obstacle, *obstacle_id* is detected for timeout amount of time.
obstacle_inside_intersection (*obstacle_id*)	Return true if the obstacle, *obstacle_id* is inside an intersection region.
Obstacle_distance_to_intersection (*obstacle_id*)	Returns distance from obstacle, *obstacle_id* to next immediate intersection.
blockage_in_safety_region (*l*, *obstacle_id*)	Returns true if the obstacle, *obstacle_id* is within the safety region of the lane, *l*. That is near an intersection region.
neighbor_lane_exist (l_1, l_2)	Returns true if there exists a neighboring lane to lane l_1. If so assign l_2 to the neighboring lane.
lane_change_allowed (l_1, l_2)	Returns true if a lane change from l_1 to l_2 is allowed. Lane change is not allowed if the lanes are separated by yellow marking or if there is a physical median between the lanes.
lane_free_of_obstacles (*l, x, y, dis*)	Returns true if the lane, *l* is free of obstacle starting from point (x_p, y_p) on the lane, *l* till distance *dis*. Here (x_p, y_p) is the projection of (*x, y*) onto lane *l*.
uturn_region_blocked (l_1, l_2, *x, y*)	Returns true if there is sufficient region on lane l_1, and l_2 surrounding the point of (x_2, y_2) for making an u-turn. Here (x_2, y_2) is projection of (*x, y*) onto lane l_2.
reach_next_cp_from (*cp, l, x, y*)	Returns true if it is possible to reach checkpoint *cp* from point (x_p, y_p) on lane *l*. Here (x_p, y_p) is the projection of (*x, y*) onto lane *l*.
get_lane (lane_id)	Returns the shape and position of the lane as a sequence of equally spaced points along the center of the lane. [(x_1, y_1), (x_2, y_2), (x_3, y_3)…]
get_position ()	Returns the position (*x, y, z*) of the AGV.
get_orientation ()	Returns the orientation (*roll, pitch, heading*) of the AGV.
get_mission ()	Returns the mission as a sequence of checkpoints. [cp_1, cp_2, .. cp_n]
get_rndf ()	Returns the route description (S, Z), refer Section 2.3 for RNDF representation.

4 Basic Capability

One of the classical approaches in solving a complex problem is to use techniques such as *hierarchical decomposition* (Russell and Norvig, Hierarchical Task Network Planning 2003). In this approach, a complex problem is divided into a hierarchy of sub-problems, with each lower layer in the hierarchy addressing the sub-problems at higher detail. The key advantage of this approach is that the complexity of the problem is reduced at all layers of hierarchy. At the higher layers, the larger problem statements are solved in more abstract steps and at lower layers the smaller sub-problems are solved at more detailed steps.

The concept of basic capability introduced in this chapter is the result of reducing the Path Planner's complexity using a hierarchical decomposition technique. The basic capabilities address the problems at lower-layer by planning a concrete path for an AGV to follow. The higher layer problem statements are addressed in steps representing which basic capability to utilize.

In the remaining of this Section: the Subsection 4.1 provides a more detailed overview of the basic capabilities, and Subsection 4.2 provides some notes on how to draw a line between basic capabilities while defining them. I then introduce Basic Planners (BPs) that provide these capabilities for the CB_PP in Subsection 4.3. Subsection 4.4 introduces the Base-Path protocol, which provides transparency between BPs to achieve easy plug-ability of new BPs. This section concludes with an overview of Zone-Navigator BP in Subsection 4.5.

In the remaining of this Section, I provide a more detailed overview on the basic capabilities in Subsection 4.1. Then provide some notes on how to draw a line between basic capabilities while defining them, in Subsection 4.2. Then I introduce the BPs that were implemented to address the UC requirement, in Subsection 4.3. To provide easy plug-ability of BPs, each BP is kept transparent of all the other BP's types. I introduce *Base-Path* protocol which provides this transparency, in Subsection 4.4. Then conclude the section with an overview of Zone-Navigator BP, which plans a path to navigate in zone in Subsection 4.5.

4.1 Overview

Urban driving requires the ability to perform a diverse set of basic-capabilities, such as following a lane, changing to a lane, and driving into a parking spot. Each such capability is achieved by a specialized planner called a Basic-Planner (BP).

A BP exhibits its basic capability by planning a path for the AGV to follow. While each BP plans a path for providing one basic-capability, a sequence (σ) of such BPs, managed by the rest of the Path Planner, plans one combined path that may be used to accomplish the complete mission.

A BP has a smaller and more specific problem statement. This enables BP to use an approach most suitable for solving that problem. For instance, a zone BP that is responsible for driving in a zone while avoiding obstacles may be implemented using a grid-based path exploration approach. On the other hand a Follow-Lane BP, responsible for following a lane, may compute the center of the lane from the lane boundaries. Each BP has the freedom to use an environment representation that is best suited for the algorithm it implements.

Table 3 lists eight BPs that address the scenarios expected at the UC. The BPs in the table are categorized based on where they operate. This classification is purely for ease of understanding.

4.2 Line between Basic Capabilities

There is some decision making required to decide where to draw a line in defining basic-capabilities. For example, is it better to have a separate Follow-Lane BP and a Change-Lane BP or is a single Follow-Road BP better? Here, a road is a collection of neighboring lanes and a Follow-Road BP could be responsible for both following a lane and changing between lanes. I use following two criteria to decide whether to have a BP for a capability A, or to divide the capability A into two sub-capabilities, B and C, and achieve each of them by two separate BPs:

<u>Criterion 1: Usage of the capability.</u> For executing any behavioral-capability, will the Re-Planner need to use BPs for B and C individually? If so it is preferred to define capabilities B and C separately and achieve each of them by separate BPs.

For example, suppose capability A is the ability to follow-road, B is to follow-lane, C is to change-lane. In the scenario where the lane being followed is blocked ahead, the Re-Planner may want to exhibit a behavioral-capability of switching to a neighboring lane, following the lane for '*d*' distance and returning back to the original lane. In order to perform this behavioral-capability, the Re-Planner needs basic-capabilities B and C individually. In this situation, the Re-Planner would be assigning the following BP sequence: BP for C (change to neighboring lane), BP for B (follow for ‘*d*’ distance) and BP for C (change to original lane). Hence, having separate BPs for B and C is preferred over having a single BP for A.

Criterion 2: Implementation of the capability. Do capabilities B and C define different problem statements? If so, each statement can be achieved more easily when separated, as compared to having a single algorithm for capability A.

For example, suppose that capability A is to drive in a zone, such as a parking-lot, and being able to park in a parking spot, capability B is to be able to drive in a zone from one point to another and capability C is to be able to park the vehicle. While zone driving (capability B) can be achieved by searching for a less precise path around obstacles at a safe distance in a large area, parking (capability C) requires searching for a highly precise path in a small area. Due to the difference in the size of the search space and the precision of the path being planned, each capability may be achieved more easily using a different approach. Hence, having them provided by two separate BPs is preferred.

4.3 Basic Planners

A Basic Planner (BP) is specialized to plan steering paths to provide a specific basic capability. A BP can be simple and straight-forward, like the Follow-Lane Basic Planner (FL_BP), which plans a path to navigate along the center of a lane. Alternatively, a BP can be complex, like the Zone-Navigator Basic Planner (ZN_BP), which plans an obstacle-free steering path through a zone. The ZN_BP requires a much more complex planner compared to FL_BP, to handle a large terrain and obstacles.

The BPs can be divided into three categories depending on where they operate, namely, on-road Basic Planners, within zone Basic Planners, and transition Basic Planners. Table 3 lists the BPs in each of these categories. They are further described below:

a) On-road Basic Planners: These BPs operate on a single road. There are four Planners in this category, namely: Follow-Lane Basic Planner (FL_BP), Change-Lane Basic Planner (CL_BP), Traffic-Lane Basic Planner (TL_BP), and U-turn Basic Planner (UT_BP).

A Follow-Lane Basic Planner plans a path to drive from one point to another on the same lane. A Change-Lane Basic Planner plans a path switch to an immediate neighboring lane that is in the same direction as the original lane. The CL_BP can only switch by one lane, so

Table 3: List of Basic Planners (BPs)

Basic Planner	*Acronym*	*Task*
	Within-road Basic Planners	
Follow-Lane	FL_BP (*wp*) FL_BP (*d*)	Plan a path to follow the center of a lane to reach the Waypoint *wp* or to travel for certain distance *d*.
Change-Lane	CL_BP (*l*)	Plan a path to change neighboring lane, *l*.
Traffic-Lane	TL_BP (*d*)	Plan a path to cover certain distance, *d* on present lane using oncoming traffic lane.
U-turn	UT_BP ($l_{next)}$)	Plan a 3 point U-turn to enter l_{next} by staying within the road with.
	Within-zone Basic Planners	
Parking	PK__BP (*pk_wp*[2])	Plan a path to park in the parking-spot specified by *pk_wp*[2].
Un-Park	UP_BP (*wp*)	Plan a path to pull out of a parking spot to face towards next waypoint *wp*.
Zone-Navigator	ZN_BP (*wp*)	Plan a path to drive through a zone avoiding static obstacles to a given waypoint *wp*.
	Transition Basic Planners	
Intersection	IT_BP (*en_wp*)	Plan a path through an intersection to reach en_wp, while considering the elements such as stop sign, traffic, and right of way.

to switch multiple lanes requires multiple instances of CL_BP. A Traffic-Lane Basic Planner plans a path to use an on-coming traffic lane to cover a certain distance and come back to present lane. The TL_BP is used when the present lane is blocked for a small distance ahead (a stalled vehicle, for instance). Finally a U-turn Basic Planner plans a path to perform a 3-point U-turn, while staying within the road boundaries. The UT_BP is required either when the road ends (a stub or dead end) or when the road is blocked and the only way out is to make a U-turn and plan an alternative route back.

b) Within zone Basic Planners: These BPs operate within a single zone, such as a parking lot. Three types of BPs operate within a zone, namely: Parking Basic Planner (PK_BP), Un-Park Basic Planner (UP_BP), and Zone-Navigator Basic Planner (ZN_BP). A Parking Basic Planner plans a path to park in a parking spot, starting from some position and orientation near the parking spot. A Un-Park Basic Planner plans a path to pull out of a parking spot and face towards a given orientation. A Zone-Navigator Basic Planner plans a path to drive to a given goal position at a given orientation and avoid any obstacles detected in the region.

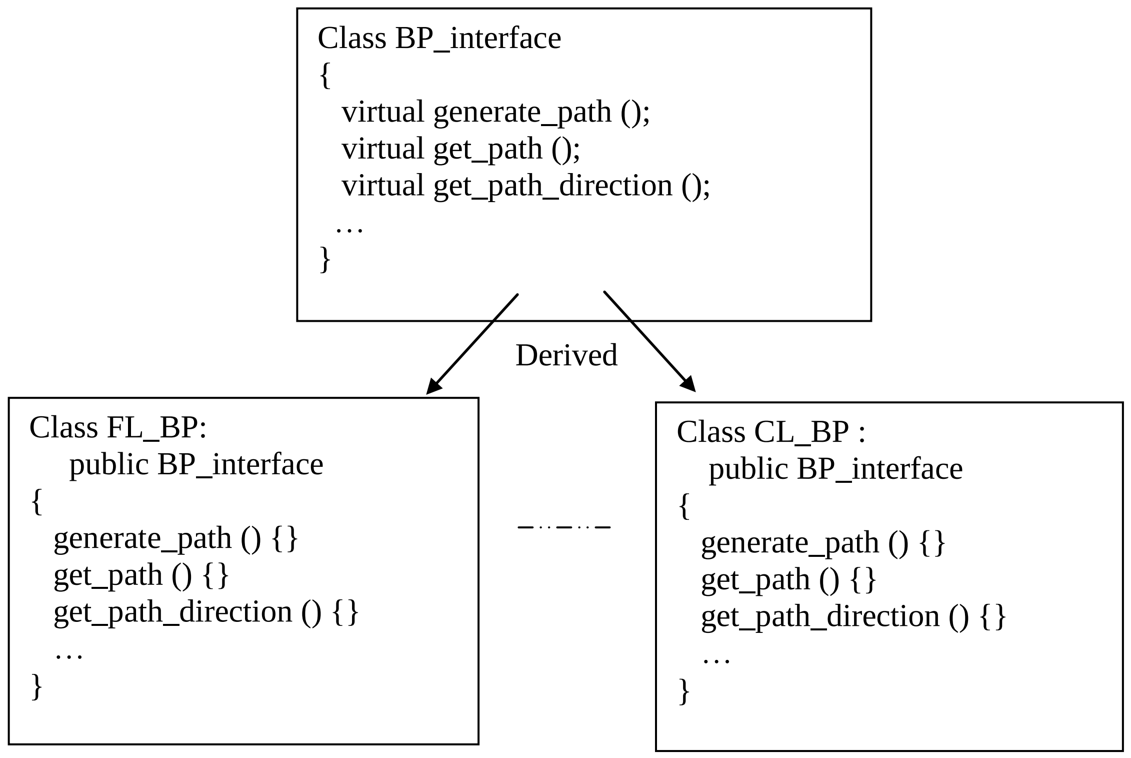
c) Transition Basic Planners: These BPs plan a path to drive from one road or zone to another road or zone. This category has only one BP, namely, the intersection Basic Planner (IT_BP). It plans a curved path starting from an exit waypoint with a given orientation to an entry waypoint with a different orientation. The ending orientation depends on whether the vehicle is entering a road or a zone. If entering a road, the ending orientation is defined to be the orientation of the new road. If entering a zone, the ending orientation is defined to be the normal to the zone’s perimeter and facing into the zone.

4.4 Plug-ability of Basic Capabilities

There will be a need to add more basic-capabilities in the future while trying to handle new scenarios such as parallel parking and traffic-jams. It is necessary to make sure of two things: a) The CB_PP should be able to use the new BPs without any major changes and, b) adding new BPs should have minimal or no effect on the existing capabilities. CB_PP architecture tries to achieve this by: having a common interface template for all BPs and by having each BP confined to plan a path for its own goal. Each of these is explained below.

1) *Common Interface*

All BPs implement a common interface template making it easier for the rest of the system to interact with any BP without knowing them specifically. Figure 8 shows the hierarchy diagram of the common interface class implemented by BPs. With this interface, the CB_PP accesses BP instances using a pointer of type BP_interface. Table 4 lists the member interface functions of BP_interface. Column one of the table lists the signature of the function, followed by column two giving a brief description of the function.



```
Class BP_interface
{
  virtual generate_path ();
  virtual get_path ();
  virtual get_path_direction ();
 …
}
```

```
Class FL_BP:
    public BP_interface
{
  generate_path () {}
  get_path () {}
  get_path_direction () {}
  …
}
```

```
Class CL_BP :
    public BP_interface
{
  generate_path () {}
  get_path () {}
  get_path_direction () {}
  …
}
```

Figure 8: Class hierarchy of BP's class

Table 4: Listing the member functions of the common interface class (BP_interface)

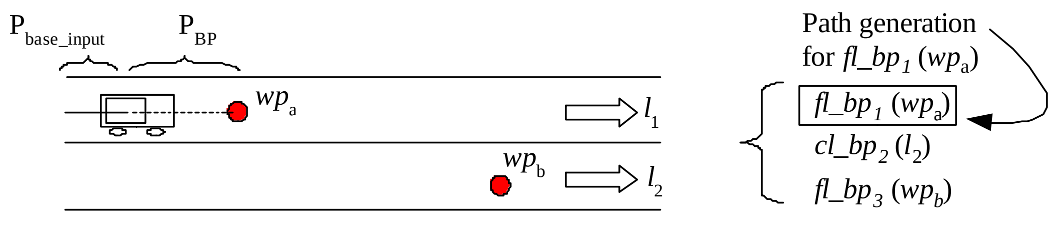
Function	Description
generate_path ($P_{\text{base\_input}}$)	Trigger the BP to generate its path
get_path ()	Return the already generated path P_{BP}
get_path_direction ()	Returns the direction of the path (*forward*/*reverse*)
is_ completed ()	Returns BP's capability is completed
get_TE_PT ()	Returns a list of traffic entry, TP_PT points on its path
get_TE_PT_path_index (k)	Returns BP's path index which represents TE_PT [k]
stop_path_generation ()	Returns if next BP in the sequence should be triggered to generate their path.
can_handle_obstacles ()	Returns true if it handles static obstacles on its path
set_path_speed (i_{path}, *speed*)	To set speed limit of BP's path
can_stop_along_the_path ()	Returns true if the AGV can stop in middle of its path
get_lane_ids ()	Returns a list of lane ids that BP's path overlays

2) *Confined Planning (Base-Path Protocol)*

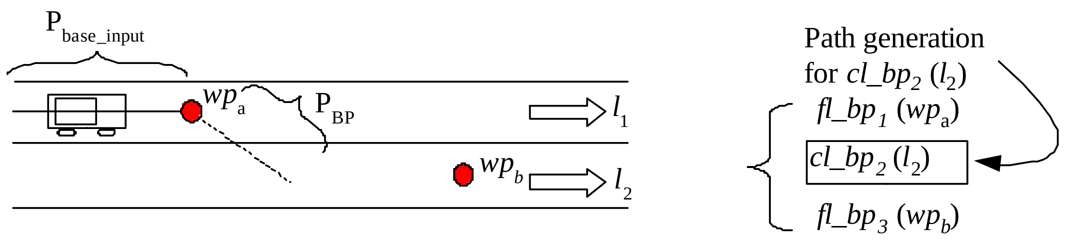
Each BP is unaware of the other BPs that it works with to plan a final path for the AGV to follow. This allows easy addition of new BPs without affecting the existing BPs. With each BP planning its own capability, it is challenging to ensure that the combined final path planned by the BP sequence (σ) is both "navigable" and "consistent over the path-planning cycles." A path is navigable if the vehicle can closely follow the path and is consistent over the path-planning cycle if the path does not have major changes between the path-planning cycles unless there is a valid cause. Each of these is achieved by the *Base-Path* protocol that is defined as follows:

a) Each BP takes an input path called the base-path ($P_{\mathrm{base\_input}}$), concatenates it with a path segment P_{BP} to achieve its own capability and then outputs the combined path $P_{\mathrm{BP\_output}} = P_{\mathrm{base\_input}} + P_{\mathrm{BP}}$. This output path ($P_{\mathrm{BP\_output}}$) from one BP then acts as an input base-path ($P_{\mathrm{base\_input}}$) for the next BP in the BP sequence.

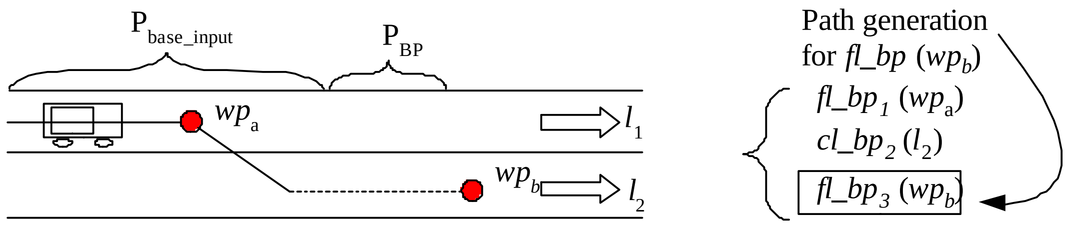
   Figure 9 illustrates this process for an example BP sequence. Here the fl_bp_1, shown in Figure 9 (a) takes the input base path $P_{\mathrm{base\_path}}$ (solid line) and extends it with P_{BP} (dotted line) to reach wp_{a}. This combined path is then used as input base path $P_{\mathrm{base\_input}}$ to cl_bp_2, as shown with solid line in Figure 9 (b). The cl_bp_2 extends the input base path with P_{BP} to switch to lane l_2. Continuing the similar process the combined path from cl_bp_2 is used as input base for fl_bp_3 to reach wp_{b}, as shown in Figure 9 (c)



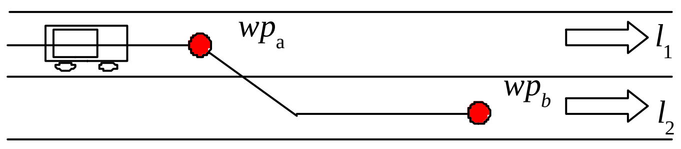
(a) $P_{\text{base\_input}}$ and P_{BP} of first BP: fl_bp_1



(b) $P_{\text{base\_input}}$ and P_{BP} of second BP: cl_bp_2



(c) $P_{\text{base\_input}}$ and P_{BP} of third BP: fl_bp_3



(d) Final output path

Figure 9: Illustration of the Base-Path protocol applied to an example BP sequence

The input base-path ($P_{\text{base\_input}}$) provides the BP with an expected "state of the vehicle" when the vehicle will start following the path it plans. The "state of the vehicle" includes the position, orientation, and rotation of the vehicle. The rotation of the vehicle implies the course of action in terms of angular turn, or the steering wheel position. Each BP is responsible for extending its $P_{\text{base\_input}}$, such that the output

$P_{\text{BP\_output}}$ is navigable. Thus the final output path from the last BP is ensured to be *navigable* by the vehicle.

b) All but the first BP in the BP sequence have a preceding BP that provides the $P_{\text{base\_input}}$. But how does the first BP get its $P_{\text{base\_input}}$? $P_{\text{base\_input}}$ is meant to provide the BP with an expected “state of the vehicle” when following its path, hence the first BP gets its $P_{\text{base\_input}}$ as the part of the path that is already being followed. That is, the $P_{\text{base\_input}}$ is formed as a part of the final path planned in the previous path-planning cycle that lies behind the present position of the vehicle. By using the part of the path previously planned to form the starting $P_{\text{base\_input}}$, the CB_PP plans a steering path that is ”consistent over path-planning cycles”.

c) What about the very first path-planning cycle where there is no previously planned path? How does the first BP get its $P_{\text{base\_input}}$? For the first path-planning cycle, a dummy $P_{\text{base\_input}}$ with two points is created for the first BP. The first point is placed at a distance ‘d_b’ behind the vehicle and the second pointis placed at the present location of the vehicle. Here, the distance ‘d_b’ is the suggested distance between the points in the path. This dummy $P_{\text{base\_input}}$ provides position and orientation of the vehicle and assumes that the system is starting with its steering wheel in the center position.

4.5 Zone-Navigator Basic Planner

The Zone-Navigator Basic Planner plans a path to drive an AGV through a zone to reach a given goal position and orientation. It plans an obstacle-free path that continues its input base path ($P_{\text{base\_input}}$) to reach a given goal state (position & orientation). The goal for a ZN_BP could be either a parking spot or an exit waypoint to leave a zone. The position and

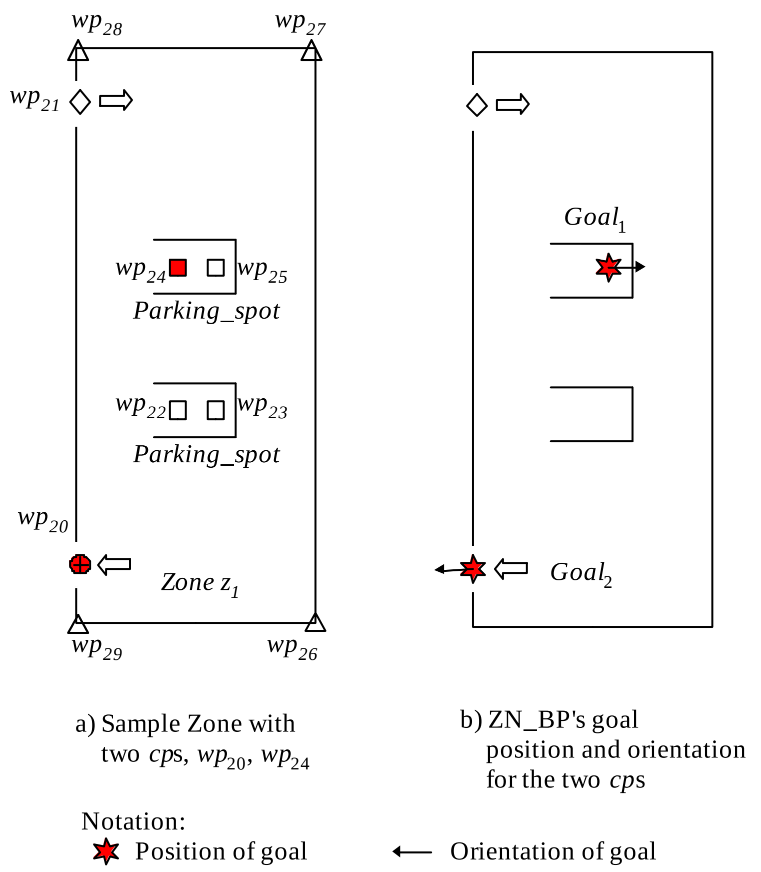


Figure 10: Position and orientation of ZN_BP’s goals

orientation of the ZN_BP’s goal are determined differently depending on the goal type. If the goal is a parking spot, then the ZN_BP’s goal position and orientation is equal to the position and orientation of the parking spot, as shown with $Goal_1$ in Figure 10. If the goal is an exit waypoint, then the ZN_BP’s goal position is equal to the position of the exit waypoint and orientation is equal to the slope of the normal to the zone’s perimeter at the exit waypoint, as shown by $Goal_2$ in Figure 10. The ZN_BP’s path planning has two parts, namely (a) Grid representation, and (b) Path Extraction.

The remainder of this section describes these two parts. I then describe a special case handled by ZN_BP when planning a path to a parking spot and then finally list two possible updates of the ZN_BP. Finally, I explain the present planner's step using an example planning scenario and evaluate its performance.

4.5.1 Grid Representation

ZN_BP uses the Gradient-Distance-Field (GDF) (Barraquand, Langlois and Latombe 1991) (Maida, et al. 2006) to provide guidance in planning a smooth path through the zone. The GDF is computed by having a grid representation covering the zone's arena. The grid has its rows and columns aligned along the GPS x-y axis to preclude the need to apply any rotations for mapping points between the GPS coordinate system and the grid coordinate system. The tile-based grid representation assigns grid space on-demand similar to a quad-tree (Finkel and Bentley 1974)(Cline and Egbert 2001) and Octree (Samet 1984). For the CB_PP implementation for UC, the grid cell was set to 32cm x 32cm.

Each grid cell maintains three binary flags to represent if the area that the cell represents is a: i) goal region, ii) zone boundary region or the region outside zone, and iii) obstacle region (Stentz and Hebert, A Complete Navigation System For Goal Acquisition in Unknown Environment 1995)(Xiaoxi and Leiting 2008). These characteristics are not exclusive, that is a grid cell can have all three flags set. Along with these three flags, each grid cell maintains three numeric attributes, namely:

i) Obstacle Penalty: Penalty given to cells near obstacle,

ii) Boundary Penalty: Penalty given to cells near the boundary, and

iii) Activation-Cost: The smallest cost of reaching a goal cell.

The goal and boundary flags for the cells are set during the initial grid setup, as this information is known at startup. Obstacle flags for the cells are set if and when static obstacles are detected. Upon setting the boundary or obstacle flags of a cell, its neighboring cells within certain distance are updated with the corresponding boundary or obstacle penalties. Finallly, the third attribute, activation cost for the cell is updated only on demand.

The boundary and obstacle flags are used to terminate the path extraction process (explained in the next step) and the goal flag is used to determine when a path extraction step has reached a goal cell. The obstacle and boundary penalty variables are used to compute the activation-cost value for the cells by providing information on how close is the cell from the nearest boundary and the obstacle. Lastly, activation-cost is used to provide heuristics for an A* based path extraction step.

4.5.2 Path Extraction

In this step, a Dubin's based path exploration is performed (Dubins 1957)(Agarwal, Prabhakar and Hisao 1995)(Scheuer and Fraichard 1996). It uses the activation-cost value computed in the previous grid representation step as a heuristic for performing an A* (Patrick 1992) based exploration of the Dubin's path exploration steps. Each node in the path exploration contains: a) position that the node represents, b) expected orientation of the vehicle if the path sequence is followed, and c) the activation-cost value of the grid cell covering the node's position. The first two values, position and orientation, help to determine if the goal is reached or to continue with the path extraction step, while the third value, activation-cost, provides the heuristics used to determine which node to explore next.

The search begins with a start node placed at the position of the last point of the input base path ($P_{\text{base\_input}}$) to the ZN_BP, the orientation value for the node is set to the angle between points $P_{\text{base\_input}}$ [n-1] and $P_{\text{base\_input}}$ [n], where ‘n’ is the size of $P_{\text{base\_input}}$, and the activation-cost value for the node is determined by mapping this start position onto the grid. The start node is then explored to determine its three children nodes, each representing the estimated position and orientation of the vehicle if it performed a maneuver of turning left, turning right, or continuing straight. The position and orientation of the children nodes are computed based on the maneuver angle and distance covered at that angle. These two values are set based on the desired granularity in the path exploration. For UC, these values are set to 10 deg maneuver angle and 1 meter distance between nodes, respectively. The activation-cost values for the nodes are determined by mapping the positions of the nodes onto the grid representation and using GDF value of the cell it maps onto.

The three children nodes are then checked if they are at a goal or if they are being blocked by a boundary or obstacle, using flags of the grid cell they map onto. If the node reached a goal cell, with an orientation close to the desired orientation, the path extraction step terminates and begins backtracking and extracting the points along the search path. If the node reaches a boundary or an obstacle cell, the node is discarded and not considered for further exploration.

Of the open new nodes not already explored, the node with least activation-cost is picked and explored next. This process is repeated until a goal is reached or all nodes are explored. If not using the grid’s representation’s GDF value as heuristic, the number of open nodes to explore would increase exponentially with the depth of the exploration tree. Thus, using the grid approach to support Dubin’s path exploration helped to extract a smooth path without the exponential tree exploration cost.

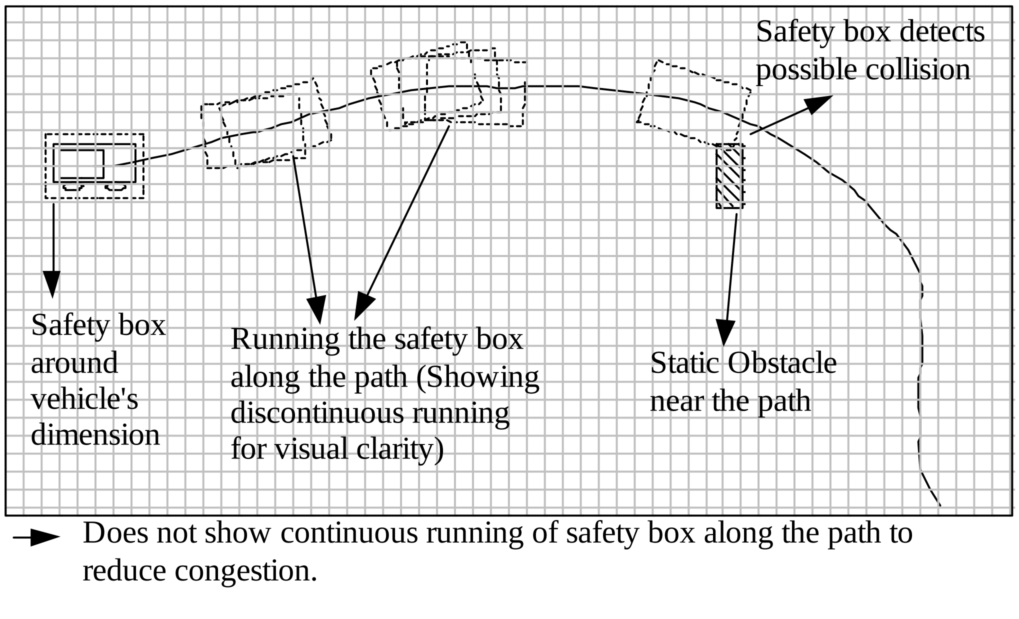
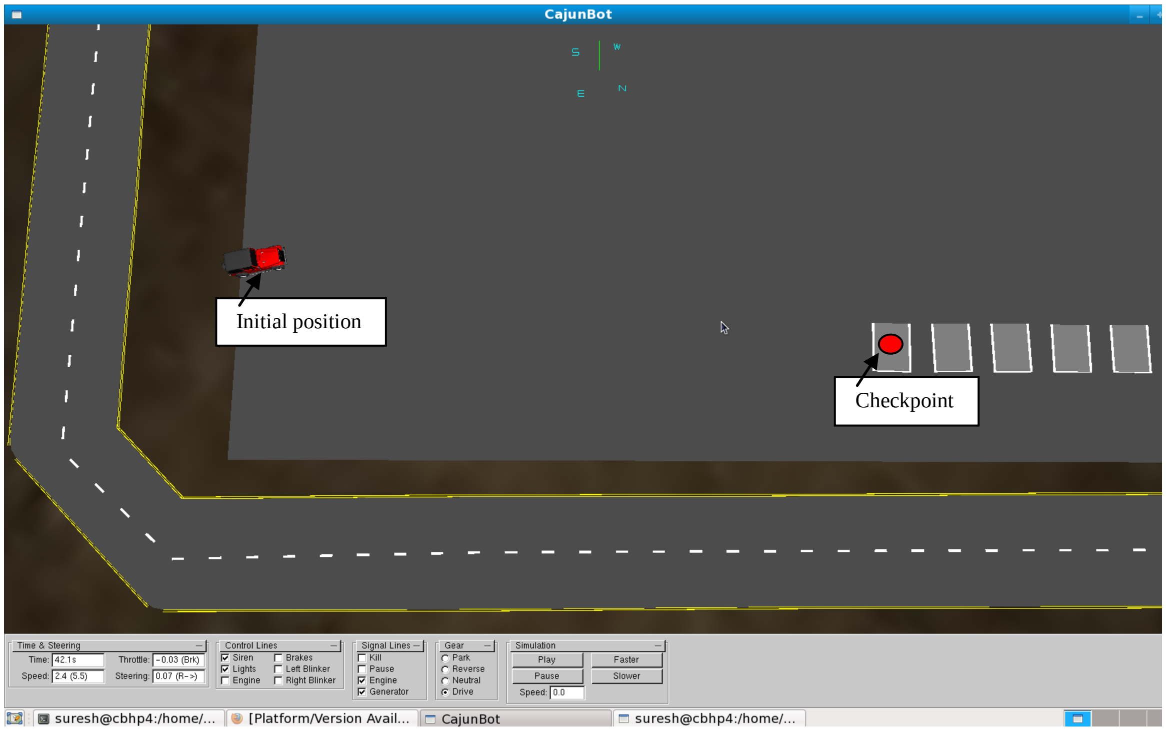

Safety box detects possible collision
Safety box around vehicle's dimension
Running the safety box along the path (Showing discontinuous running for visual clarity)
Static Obstacle near the path
Does not show continuous running of safety box along the path to reduce congestion.

Figure 11: Illustration of safety box walking through a ZN_BP's path for safety check

Static obstacles are registered and used in computing the activation-cost values of the cells, but this is not sufficient to ensure that it is safe to follow the path being extracted, since the vehicle is not a point object and has some length (*Len*) and width (*Wid*) along its body. Therefore, during path extraction the safety of the path being extracted is assured by running a safety box along the path and checking if any cells containing static obstacles fall within the safety box region, as shown in Figure 11. The safety box is oriented along the estimated orientation of the vehicle at that point. The length (*ln*) and width (*wd*) of the safety window are chosen to be ($Len + \varepsilon_l$) and ($Wid + \varepsilon_w$) respectively, where ε_l and ε_w are the required safety distances along and on the sides of the vehicle. Running this safety window in small intervals greatly covers nearby obstacles at turns.



Time: 42.1s
Throttle: -0.03 (Brk)
Speed: 2.4 (5.5)
Steering: 0.07 (R->)
[✓] Siren
[ ] Brakes
[✓] Lights
[ ] Left Blinker
[ ] Engine
[ ] Right Blinker
[ ] Kill
[ ] Pause
[✓] Engine
[✓] Generator
Park
Reverse
Neutral
[✓ Drive]
Speed: 0.0

Figure 12: Example zone planning scenario to reach a parking spot

4.5.3 Example Scenario

This section explains the step followed by the Zone-Navigator using an example planning scenario. Figure 12 shows the planning scenario with estimated start position of the vehicle and a goal parking spot. The outer grey strip region represents a pair of lanes surrounding the central grey zone area. The zone has five parking-spots, out of which the closest parking-spot, at a distance of 54 meters, as shown in the figure is specified as the checkpoint to reach. The planner needs to plan a path to park the AGV in this parking-spot.

At the end of this section I compare the performance of the Dubin's path extraction process performance when using grid-representation's GDF value as heuristic, when using Euclidean distance as heuristics and when not using any heuristics.

Figure 13 shows the first step of ZN_BP, where a grid is overlaid to cover the region of interest. The outer solid-red region represents the cells marked as boundary region. The solid-red region near parking-spots represents cells marked as obstacle regions. These phantom obstacle regions are marked to guide the planner for proper entry into the parking-spot. Figure 14 shows the grid representation with the GDF values computed. The cells are cyclic-color coded based on their GDF value.

Figure 15 shows the results of the Dubin's based path exploration process using the GDF values as the heuristic. The exploration processes expanded 192 nodes, before finding the navigable path at a tree depth of 63. Figure 16 shows the path explored without the grid display.

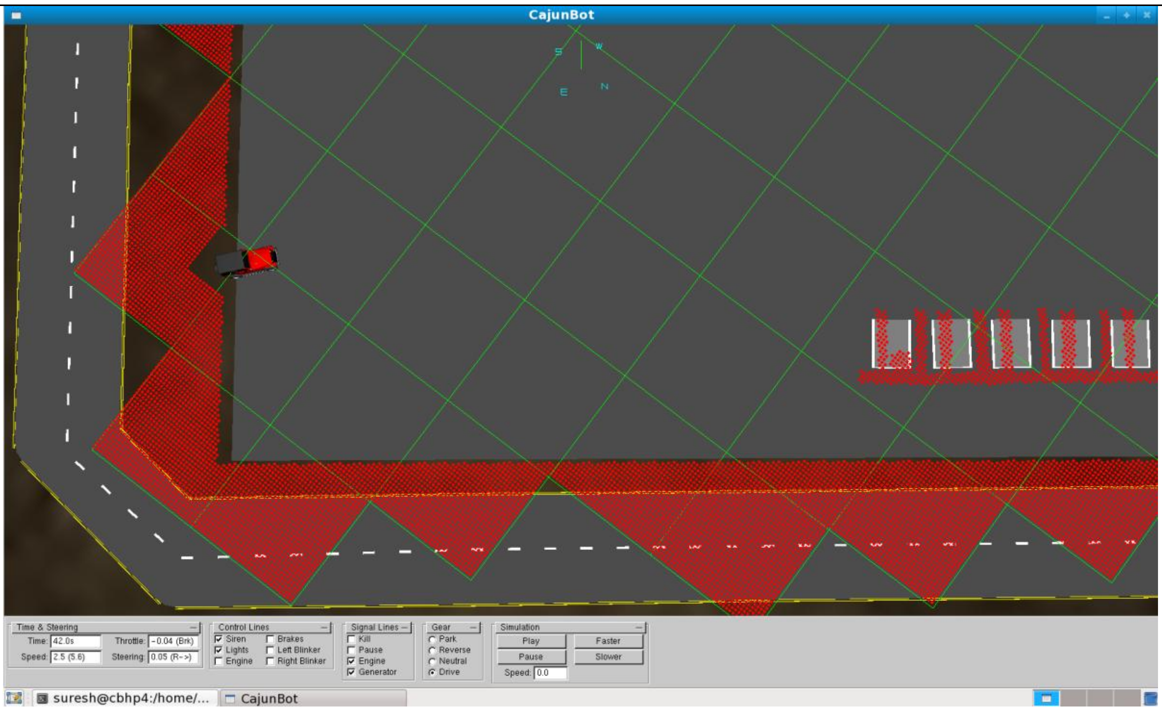
Figure 17 shows the same Dubin's based path exploration process using the Euclidean distance to goal as the heuristic for tree exploration. For the present scenario, the exploration process expanded 163,788 nodes before finding the navigable path to the parking-spot.

Dubin's based path exploration step showed better performance when using GDF value

Table 5: Performance comparison between different heuristics for Dubin's based path exploration for Figure 12 scenario

	GDF as heuristic	Euclidean distance to goal as heuristic	No heuristics
Performance: Number of nodes explored	192	163,788	$O\ (3^d)$, where d= 54
Performance: Computation time on Intel (R) core 2.00 GHz machine	200 ms	7.2 sec	~

when compared to using Euclidean distance. This is because GDF values take into account the distance to the goal and the also the obstacles in the region. While, Euclidean distance heuristics takes into account only the distance to the goal position. Table 5 lists the above result including estimated performance when no heuristics are used. With no heuristics, the average performance for breadth-first or depth-first search is in the order of $O\ (3^{d})$, where d is the depth of the exploration tree. In the present scenario with goal being at a distance of 54m from start position and path exploration being performed in steps of 1m, d=54. The second column in the table list the computation time when executed on an Intel (R) core 2.00 GHz machine.



CajunBot

Time & Steering: Time: 42.0s Throttle: -0.04 (Brk) Speed: 2.5 (5.6) Steering: 0.05 (R->)

Control Lines: [✓ Siren] [✓ Lights] Engine Brakes Left Blinker Right Blinker

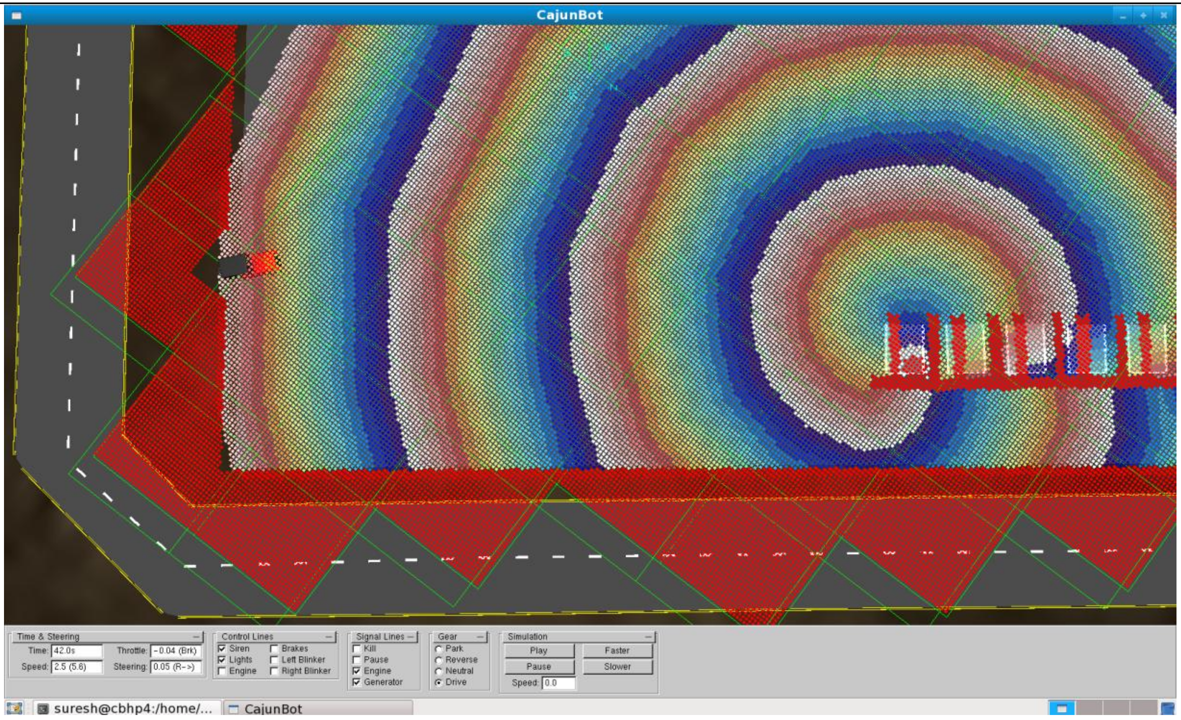
Signal Lines: Kill Pause [✓ Engine] [✓ Generator]

Gear: Park Reverse Neutral [✓ Drive]

Simulation: Play Faster Pause Slower Speed: 0.0

suresh@cbhp4:/home/... CajunBot

Figure 13: Grid representation with boundary and simulated obstacles regions



CajunBot

Time & Steering: Time: 42.0s Throttle: -0.04 (Brk) Speed: 2.5 (5.6) Steering: 0.05 (R->)

Control Lines: [✓ Siren] [✓ Lights] Engine Brakes Left Blinker Right Blinker

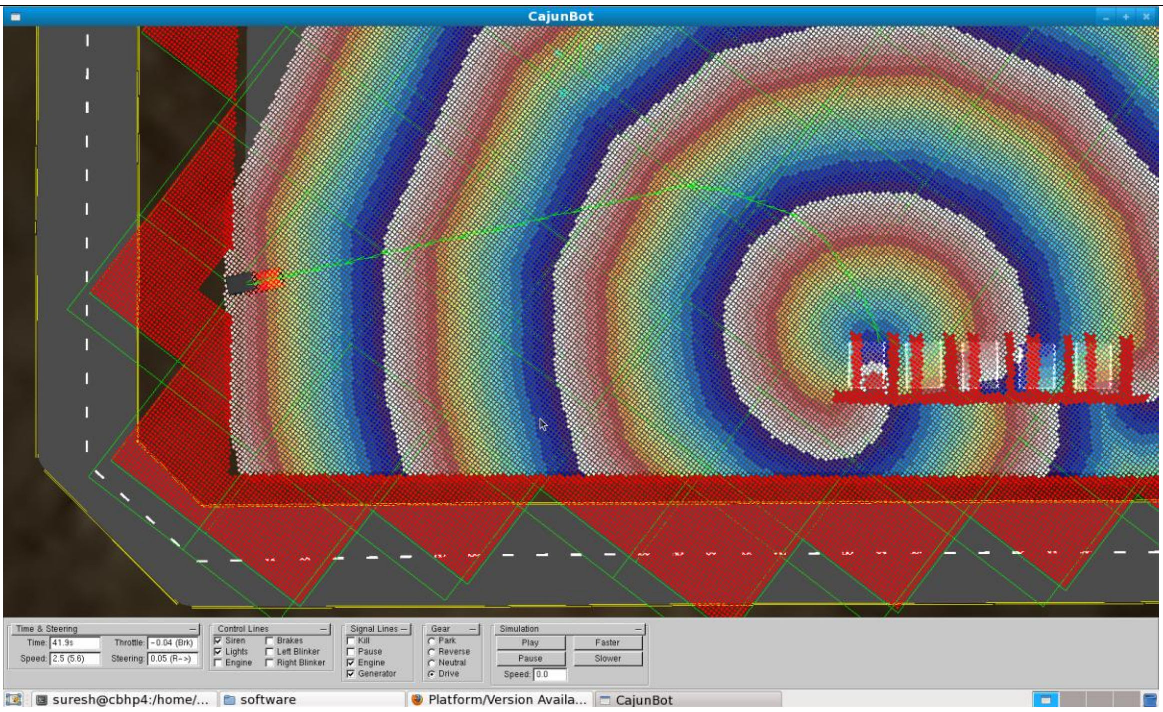
Signal Lines: Kill Pause [✓ Engine] [✓ Generator]

Gear: Park Reverse Neutral [✓ Drive]

Simulation: Play Faster Pause Slower Speed: 0.0

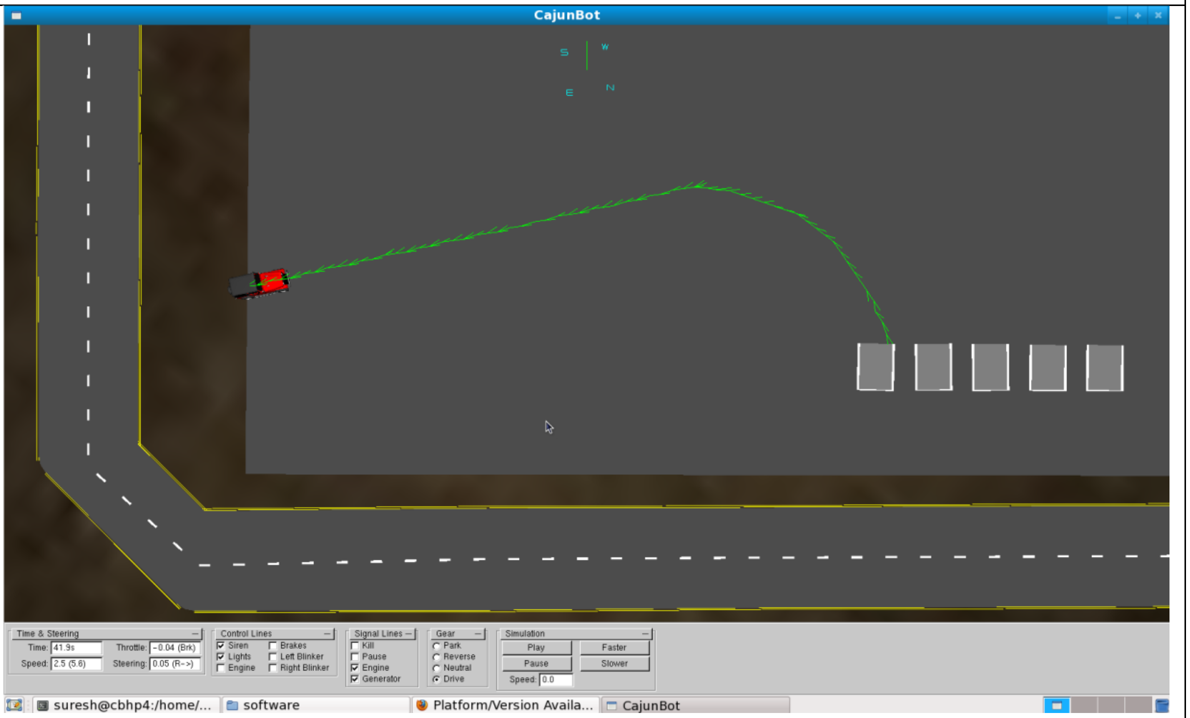
suresh@cbhp4:/home/... CajunBot

Figure 14: Grid representation with GDF values to reach the goal position



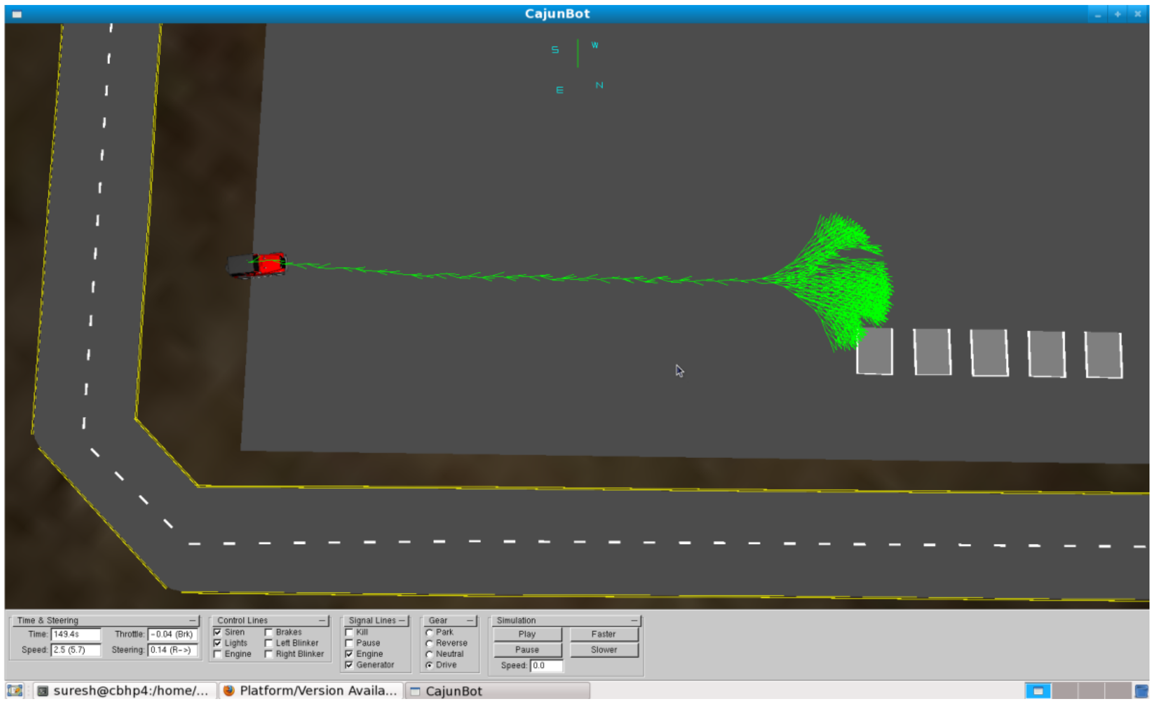
Time: 41.9s Throttle: -0.04 (Brk)
Speed: 2.5 (5.6) Steering: 0.05 (R->)
[✓ Siren] [✓ Lights] Engine Brakes Left Blinker Right Blinker
Kill Pause [✓ Engine] [✓ Generator]
Park Reverse Neutral [✓ Drive]
Speed: 0.0

Figure 15: GDF grid values along with Dubin's based path extraction



Time: 41.9s Throttle: -0.04 (Brk)
Speed: 2.5 (5.6) Steering: 0.05 (R->)
[✓ Siren] [✓ Lights] Engine Brakes Left Blinker Right Blinker
Kill Pause [✓ Engine] [✓ Generator]
Park Reverse Neutral [✓ Drive]
Speed: 0.0

Figure 16: Dubin's based path extraction using GDF as heuristics



Time & Steering — Time: 149.4s; Speed: 2.5 (5.7); Throttle: -0.04 (Brk); Steering: 0.14 (R->)

Control Lines — ✓ Siren; ✓ Lights; ☐ Engine; ☐ Brakes; ☐ Left Blinker; ☐ Right Blinker

Signal Lines — ☐ Kill; ☐ Pause; ✓ Engine; ✓ Generator

Gear — Park; Reverse; Neutral; [✓ Drive]

Simulation — Speed: 0.0

Figure 17: Dubin's based path extraction using Euclidian distance as heuristics

4.5.4 Special Case in ZN_BP

As mentioned earlier the ZN_BP can have a parking spot or a zone exit waypoint as its goal. If the goal is a parking spot, then the path extracted by the path extractor is truncated at certain distance '$d_{parking}$' from the end. This is performed so that the ZN_BP provides a path to drive the AGV close to the parking spot at an orientation from which the Parking Basic Planner (PK_BP) can take over and park the vehicle using a more accurate path.

In the initial iterations of development of ZN_BP, the path generated by the ZN_BP was used to pull all the way into the parking spot. The vehicle could park in the spot, but the final position and orientation of the vehicle were not within the desired tolerance. This is because of the lower resolution of the path exploration by ZN_BP. This issue could be addressed by reducing the maneuver angle or reducing the distance covered by the children nodes during the path exploration. But this again would lead to high search space and time. Sticking to the basic principle of the CB_PP's architecture (refer Section 4.2), it is easier to divide the responsibilities among different specialized planners to keep the planning system simple. Hence, I let the Parking Basic Planner take responsibility for parking the vehicle.

4.5.5 Possible Updates to ZN_BP

There are two drawbacks for the present ZN_BP, namely, insufficient path extraction heuristics, and inaccurate path exploration steps. Each of these is explained as follows.

(a) Insufficient path extraction heuristic: While there are two parameters that define if a goal is reached, namely goal position and goal orientation, only one parameter, the distance from the goal, is used as the heuristic guide. This sometimes will misguide the path exploration process to reach the goal position in a wrong orientation. The present system does suffer from this drawback, leading to increased search space and back tracking especially near the goal region. Using improved heuristics that takes into account both the position and orientation requirement will improve the computational efficiency of the path extraction process.

(b) Inaccurate path exploration steps: The present path exploration step assumes constant speed for the vehicle while determining the position and orientation of the next search nodes. A vehicle may not travel at a constant speed and/or turning angle. Simulating more accurate vehicle speed and maneuvering can lead to more accurate simulation of the vehicle's trajectory.

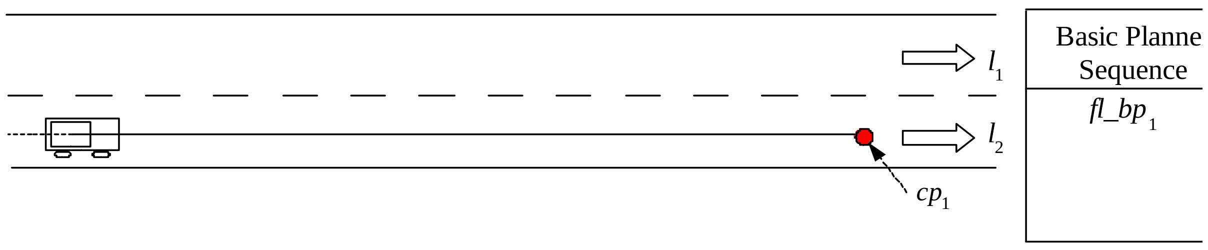
5 Behavioral Capability (Re-planner)

Due to the dynamic nature of the environment and lack of complete-knowledge of the track, it is rarely possible to plan a mission that can be used without any modifications over the course of the run. The change of plan that is required to handle a new situation encountered during the course of mission, defines the behaviors of the Path Planner. The ability to change the existing plan to handle such situations is called the *Behavioral Capability.* The present section introduces these behavioral capabilities in more detail.

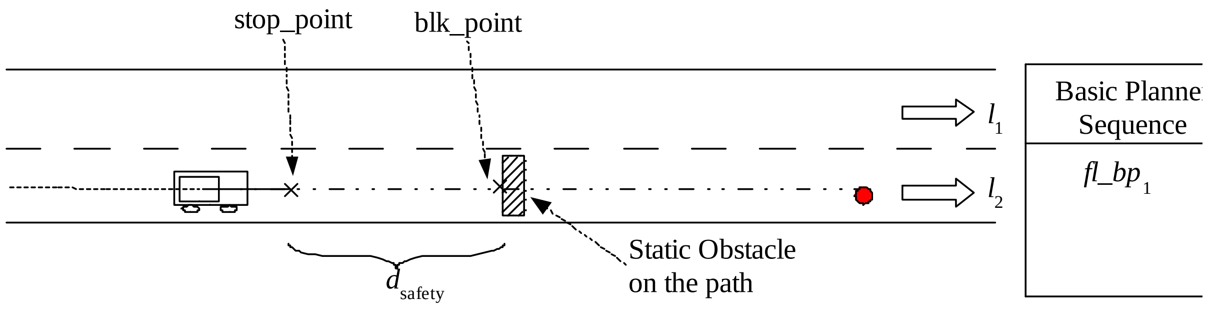
The following subsection provides an overview of the behavioral capabilities of the CB_PP. Subsection 5.2 explains how plug-ability of behavioral capability is achieved. Finally, section 5.3 lists some possible updates to handle dynamic obstacles by changing a plan and to handle situations encountered within a zone.

5.1 Overview

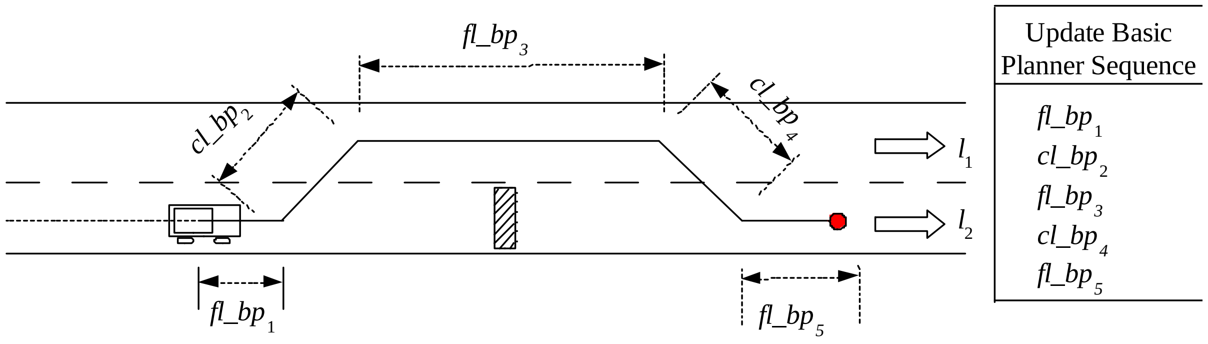
As an AGV executes a plan, there are many situations that can be encountered that may require changing the plan. For example, there could be a blocked lane or a blocked intersection, each of which making it impossible to continue with the existing plan. In case the lane is blocked, to get around the blockage, the AGV may switch to a neighboring lane or may use an on-coming traffic lane. The capability to perform one such action is called a behavioral-capability. Behavioral-capabilities are performed by modifying the existing Basic Planner (BP) sequence to add or remove BPs, or communicating with existing BPs to change their goal or changing the speed limits of the already planned path. Each behavioral-capability is implemented by a separate planner, called a Behavioral-Planner (BhP).



(a) Initial BP sequence to reach checkpoint cp_1.



(b) Detecting a static obstacle and modifying the path speed to maintain safety distance d_{safety}.



(c) Avoiding the static obstacle by switching to neighboring lane.

Figure 18: Example scenario showing Re-planner module handling a lane blockage

Figure 18 describes an example scenario of a lane blockage and illustrates how a situation can be handled by modifying the BP sequence. As an initial situation a fl_bp_1 was being used to follow a lane to reach cp_1, as shown in Figure 18 (a). Upon encountering a static obstacle along the path, the Supervisor module modifies the speed limits on the existing path so as to maintain a minimum safety distance from the obstacle, shown in Figure 18 (b). Then the static obstacle situation is handled by first changing the goal of *fl_bp*$_1$ to navigate till ‘stop point’, then updating the BP sequence to switch to a neighboring lane *l*$_1$ using *cl_bp*$_2$, follow the new lane for certain distance using *fl_bp*$_3$, switch back to the original lane *l*$_2$ using *cl_bp*$_4$, and finally continue on lane *l*$_2$ to reach *cp*$_1$ using *fl_bp*$_5$. Figure 18 (c) shows the new modified BP sequence.

The decision rules on when and which BhP should be utilized is represented as a deterministic state-machine, as shown in Figure 19. Here, the states represent either the CB_PP’s belief of what situation the AGV is in (‘safe’ state, ‘confirm_obstacle’ state), or represents the behavioral capability being exhibited using a BhP (‘use_neighboring_lane’ state, ‘use_traffic_lane’ state).

The states in the state-machine are connected via the edges that represent the conditions for exhibiting a behavioral capability or for changing the system’s belief about the AGV’s situation. The BhPs together with the state-machine is called the Re-Planner module.

Table 6 lists the states in the state machine designed to handle possible situations at UC. The first column lists the names of the states, second column briefly describes the states and the third column describes the actions performed by each state.

Table 7 describes the state transitions of the state-machine. The first column lists the starting state, second column lists the condition to be satisfied to transition to the state mentioned in third column. The functions used in the condition column are provided by the World_State as listed in Table 2, except for functions *obstacle_on_path* (*obstacle_id*), *no_obstacle_on_path* () and *bp_in_sequence* (*bp*, σ).

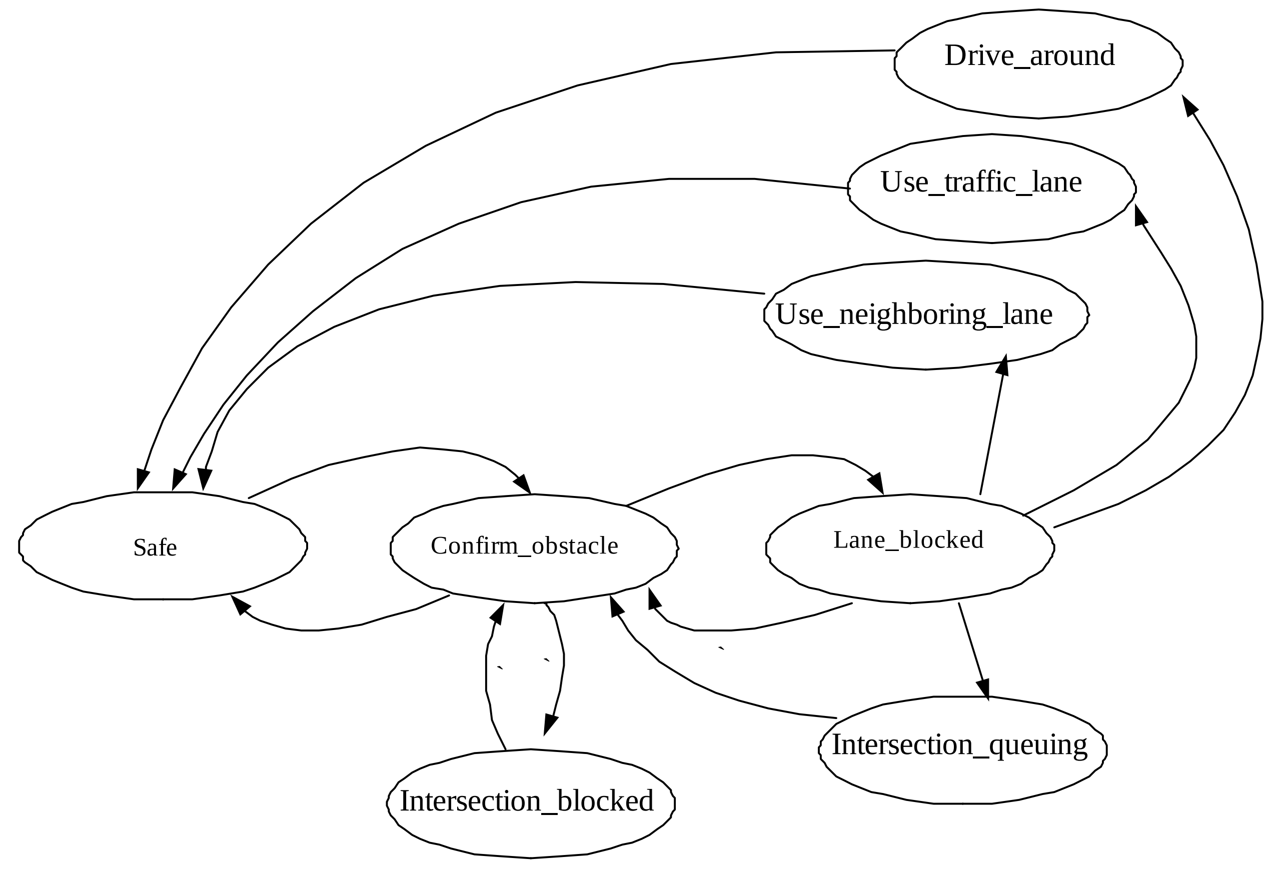


Figure 19: Re-planner's state-machine representation

Here *obstacle_on_path* (*obstacle_id*) returns true if in the present path-planning cycle, the Supervisor module informed of obstacle with id *obstacle_id*, to be the closest obstacle on the path. Function *no_obstacle_on_path* () returns true if no obstacles are detected on the path. And function *bp_in_sequence* (*bp*, *σ*) returns true if the Basic Planner *bp*, has not completed its capability and is still in the *bp* sequence *σ*.

The state machine starts at a default 'safe' state. Upon detecting an obstacle with an id *obstacle_id* on the path, the Supervisor module informs the Re-planner. Upon being informed of the obstacle on the path, the state machine transits to 'Confirm_obstacle' state. Here, the Re-planner confirms the obstacle for certain *timeout* and is from a comfortable distance of *reaction_distance*. At which time it confirms the obstacle, notes its id as *confirmed_obstacle_id* and transits to 'Lane_blocked' state, if the obstacle is not within any intersection region. If the obstacle is within an intersection region, the state-machine transits to 'Intersection_blocked' state.

The lane on which the obstacle is detected is noted as l_1. From 'Lane_blocked' state, the state-machine makes a decision to wait and perform intersection queuing behavior if the obstacle is within the *queuing_distance* from an intersection. Else, if the blockage is not within the safety region of lane l_1, there exist a neighboring lane l_2, in the same direction as lane l_1, with lane change allowed from l_1 to l_2 and lane l_2 is free of obstacle for $safety\_distance_{change\_lane}$) then state-machine transits to 'Use_neighboring_lane' state. Here, a safety region is a region near an intersection within which a lane-change operation is not allowed. A lane change between a pair of lanes may not be allowed if the lanes are separated by yellow marking or if there exists a physical median between the lanes. And $safety\_distance_{change\_lane}$ is the minimum distance on lane l_2 that is required for using the lane for covering the detected obstacle.

From 'Lane_blocked' state, if the blockage from obstacle is not within the safety region, there exist an oncoming traffic lane which is free of obstacle for $safety\_distance_{traffic\_lane}$, then state-machine transits to 'Use_traffic_lane' state.

From 'Lane_blocked' state, if the on-coming lane, l_2 too is blocked within $safety\_distance_{traffic\_lane}$ distance, and there is enough free-space for performing u-turn and the next checkpoint in the mission is reachable from l_2, then state-machine transits to 'Drive_around' state.

From ‘Lane_blocked’ state, if the obstacle with id, *confirmed_obstacle_id* is not detected to be the closest obstacle on the path by the Supervisor module, then the state-machine transits back to ‘Confirm_obstacle’ state. The state-machine transits from ‘Use_neighboring_lane’, ‘Use_traffic_lane’ or ‘Drive_around’ state to ‘Safe’ state if all the *bp*s added to the BP sequence, have completed their capabilities and are removed from BP sequence.

Table 6: List of states in the Re-planner’s state machine representation

State	*Semantics*	*Actions*
Safe	Normal execution (default state)	No action.
Confirm_obstacle	Obstacle detected but not confirmed	Continuously evaluation obstacle.
Lane_blocked	Confirmed the detected obstacle	Check state transition table to jump to a state and take action.
Intersection_queuing	Obstacle is just before an intersection and so perform intersection queuing behavior	No action, just wait queuing. Can extend this state to have a timer and perform a different action if queuing for really long time.
Intersection_blocked	Obstacle is within intersection region	No action, just wait for intersection to get clear.
Use_neighboring_lane	Use neighboring lane within the same road to pass around obstacle	Modify BP sequence to switch to neighboring lane of the same road to cover the blockage and return back to original lane. That is add [cl_bp_2, fl_bp_3, cl_bp_4, fl_bp_5,] to σ, refer Figure 18.
Use_traffic_lane	Use on-coming traffic lane to pass obstacles	Modify BP sequence to use oncoming traffic lane to cover the blockage and return back to original lane. That is add [*tl_bp*] to σ.
Drive_around	Complete road is blocked, need to take an U-turn and find path to the goal.	Register road blockage, use UT_BP to switch to oncoming traffic lane and determine a sequence of BPs to complete the remaining mission. That is add [*ut_bp*] to σ.

Table 7: State transition table of the Re-Planner module

(Here, *obstacle_id* is the closest obstacle on the path detected in the present path planning cycle.)

Present state	*Condition*	*New state*
Safe	*obstacle_on_path* (*obstacle_id*)	Confirm_obstacle
Confirm_obstacle	*no_obstacle_on_path* ()	Safe
Confirm_obstacle	*obstacle_detected_for_time* (*obstacle_id*, *timeout*) && *distance_to_obstacle* (*obstacle_id*) < *reaction_distance*	Lane_blocked *confirmed_obstacle_id = obstacle_id*
Confirm_obstacle	*obstacle_detected_for_time* (*obstacle_id*, *timeout*) && *distance_to_obstacle* (*obstacle_id*) < *reaction_distance* && *obstacle_inside_intersection* (*obstacle_id*)	Intersection_blocked
Lane_blocked	*obstacle_distance_to_intersection* (*obstacle_id*) < *queuing_distance*	Intersection_queuing
Lane_blocked	//l_1: lane with blockage *!blockage_in_safety_region* ($l_{1,}$ *obstacle_id*) && *neighbor_lane_exist* (l_1, l_2) && l_1*.direction* = l_2*.direction &&* *lane_change_allowed* (l_1, l_2) && *lane_free_of_obstacles* (l_2, *obstacle.x*, *obstacle.y*, *safety_distance*$_{change\_lane}$)	Use_neighboring_lane add [*cl_bp*$_2$, *fl_bp*$_3$, *cl_bp*$_{4,}$ *fl_bp*$_5$,] to *σ*
Lane_blocked	//l_1: lane with blockage *!blockage_in_safety_region* ($l_{1,}$ *obstacle_id*) && *neighbor_lane_exist* (l_1, l_2) && l_1*.direction* != l_2*.direction* && *lane_free_of_obstacles* (l_2, *obstacle.x*, *obstacle.y*, *safety_distance*$_{traffic\_lane}$)	Use_traffic_lane add [*tl_bp*] to *σ*
Lane_blocked	//l_1: lane with blockage *!blockage_in_safety_region* ($l_{1,}$ *obstacle_id*) && *neighbor_lane_exist* (l_1, l_2) && l_1*.direction* != l_2*.direction* && *!lane_free_of_obstacles* (l_2, *obstacle.x*, *obstacle.y*, *safety_distance*$_{traffic\_lane}$) && *!uturn_region_blocked (*l_1, l_2, *obstacle.x*, *obstacle.y) &&* *reach_ cp_from* (*cp*, l_2, *obstacle.x*, *obstacle.y*)	Drive_around add [*ut_bp*] to *σ*, call HLP's to re-plan remaining mission.
Lane_blocked	*confirmed_obstacle_id != obstacle_id*	Confirm_obstacle
Intersection_blocked	*confirmed _obstacle_id != obstacle_id*	Confirm_obstacle
Intersection_queuing	*confirmed _obstacle_id != obstacle_id*	Confirm_obstacle
Use_neighboring_lane	*!bp_in_sequence* (*cl_bp*$_{4,}$ *σ*)	Safe
Use_traffic_lane	*!bp_in_sequence* (*tl_bp*, *σ*)	Safe
Drive_around	*!bp_in_sequence* (*ut_bp*, *σ*)	Safe

5.2 Plug-ability of Behavioral Capability

For easy addition of new behavioral-capabilities without affecting the existing ones, each behavioral-capability is represented as a separate state in the deterministic state-machine. In this representation, adding a new behavioral-capability requires adding a new state representing the capability to the state-machine. A state is added to the state machine by connecting it to the relevant states via edges representing the conditions when the capability can be utilized. Adding a state to the state-machine does not affect the state-transition of the existing states, except for the case when the conditions for transiting to the new state satisfy along with the conditions for transiting to another state. This leads to introducing non-determinism to the state-machine. Such transition conflicts may be resolved by prioritizing the state transitions from a state.

5.3 Possible Updates

The present Re-planner system is designed to address the situations expected during the Urban Challenge, but may be updated to handle more situations. Two such possible updates the the Re-planner module are listed below.

5.3.1 Handling Dynamic Obstacles by Changing the Plan

The present system will not change the Basic Planner (BP) sequence or the goal of any BP inorder to handle a dynamic obstacle encoutered along its path. This is usually not required as in most cases the disturbance created by a dynamic obstacle is temporary. However, there may be situations where it is preferred to change the BP sequence or change the goal of a BP in the sequence upon detecting a dynamic obstacle. For example, instead of following a slow moving vehicle for a long time it may be preferred to change the lane and pass the leading vehicle.

The present CB_PP architecture can be updated to exhibit such behavior by modifying the Supervisor module to check for both static and dynamic obstacles along the path and inform Re-Planner before it makes any decisions. The Re-Planner module's state machine needs to be updated to handle dynamic obstacles along the path. For example, in the current scenario, the Re-Planner module can add a CL_BP to the BP sequence if a dynamic obstacle shows the characteristics of slow moving traffic.

5.3.2 Re-planner to Handle Situations within a Zone

In the present version of CB_PP, the Re-Planner module does not handle any situations encountered within a zone. The Zone-Navigator Basic Planner (ZN_BP) is expected to handle all its situations encountered locally. This may not always be a good separation as there can be situations that a ZN_BP cannot handle within its domain and requires re-planning. As an example, consider the situation where the area around an exit waypoint of a zone is blocked. The ZN_BP, trying to reach this waypoint in order to exit the zone, cannot locally handle the situation. It requires a different exit from the zone and such decisions should be performed by the Re-Planner module.

An un-passable lane situation is informed to and handled by the Re-planner, un-passable zone exits can be informed to the Re-planner. The Re-planner should search for an alternative exit waypoint that allows continuing with the mission. The present system does not check for un-passable zone regions, leading to an indefinite waiting of AGV within the zone.

6 High-Level Planner

This section introduces the High-Level Planner (HLP) which determines the sequence of Basic Planners (BPs) that can achieve a given mission (MDF). Figure 20 shows the black box diagram of the HLP module. It takes as input the route description (RNDF), the mission to accomplish and outputs the BP sequence.

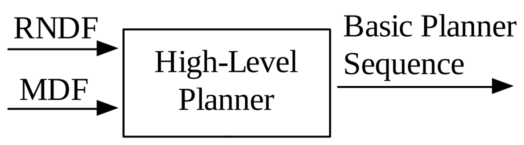


Figure 20: A black box diagram of the HLP module

HLP plans the BP sequence in three steps as shown in Figure 21. In the first step, the Graph Creator represents the urban environment description as a directed graph, known as the *rndf_graph*. In the next step the rndf_graph is searched to find a sequence of logically connected sub-goal waypoints (*SWP*) that will guide the AGV to reach all the MDF checkpoints. This sequence of *SWP*s is called the Mission Plan. This step is performed by the Mission Planner. Finally, the BP Extractor module plans a sequence of BPs that can drive through all the sub-goal waypoints. Each of these three modules is discussed in-detail in the following subsections.

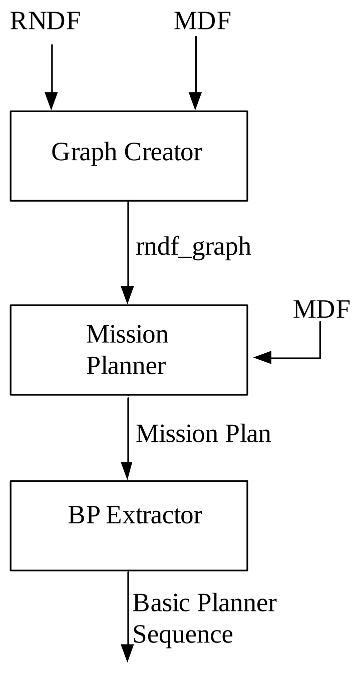


Figure 21: Dataflow diagram of the High-Level Planner (HLP) module

Figure 22 illustrates the terminology used in the

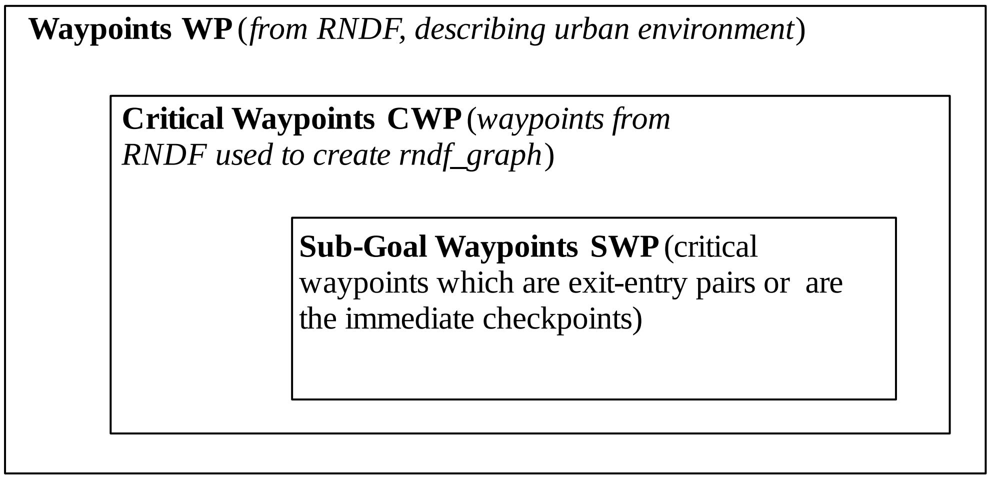

Waypoints WP (*from RNDF, describing urban environment*)
Critical Waypoints CWP (*waypoints from RNDF used to create rndf_graph*)
Sub-Goal Waypoints SWP (critical waypoints which are exit-entry pairs or are the immediate checkpoints)

Figure 22: Waypoint terminology used in HLP

remainder of the section. The input RNDF describes the urban environment using the waypoints (*WP*s). A filtered subset of these *WP*s, called critical waypoints (*CWP*s), form the rndf_graph. And a subset of these critical waypoints (from rndf_graph) forms the sub-goal waypoints (*SWP*s). A sequence of these sub-goal waypoints is called a Mission Plan.

$$SWP \subseteq CWP \subseteq WP$$

The rest of section describes each of the three steps of HLP in detail. The final step which extracts the BP sequence also presents how a logic-based planning approach can be used to provide easy plug-ability of new BPs. It then describes the decision-rules based alternative approach for extracting BP sequence that was implemented and tested during the UC.

6.1 Graph Creator

The Graph Creator converts the description of the urban environment provided in an RNDF to a directed rndf_graph representation. The RNDF description includes a list of segments, where a segment is a collection of lanes, the description of lanes as a connected sequence of waypoints, and a description of a list of zones as free-travel areas with possible parking spots inside them. RNDF also provides connectivity between all these different units (lanes and zones) and provides information about stop sign locations.

The rndf_graph is represented as a directed graph (V, E), where V the set of vertices are the critical waypoints (CWP) from RNDF. And the set E of edges represents the connection between the vertices V. Mathematically representing:

Rndf_graph = (V, E)

$V = CWP$

$E \subseteq V \times V$

$CWP = CP \cup EnWP \cup ExWP$

Critical waypoints (CWP): CWP are the waypoints from RNDF that are either specified as CheckPoints *CP* in the MDF mission or are listed in the Exit Waypoint (*EX_WP*) list or the Entry Waypoint (*EN_WP*) list. In this document CWP is used as the abbreviation of Critical waypoints, while *CWP* is used to represent a set of critical waypoints.

The waypoints that are not counted as critical waypoints are considered to be non-critical waypoints (*NCWP*).The *NCWP*s are provided in the RNDF to represent the shape of the lane and are later used by the Path Planner to plan the actual path to be followed.

$NCWP = WP - CWP$

Edges: Directed edges E between the vertices V represent the connections between these *CWP*. The edges are annotated with the estimated travel time between the *CWP*s. The travel time is computed using the distance between the *CWP*s and the speed limit for the area (road, zone or intersection). This annotated time includes any extra penalty for slowing down, such as at intersections. The rndf_graph with time-annotated edges is used to search for a path to all of the MDF checkpoints in the shortest time.

The rndf_graph is created from RNDF in following five steps:

a) Filter out set of critical waypoints *CWP* from RNDF waypoints using the following pseudo code:

```
for all $wp_i$ in WP
begin
  if ($wp_i.cp$ || $wp_i.en\_wp$ || $wp_i.ex\_wp$)
  then
     $CWP$.insert ($wp_i$)
  endif
end
```

b) Represent each waypoint wp_i in *CWP* as a vertex in the rndf_graph.

c) Create edges for vertices within each road.

d) Create edges for vertices within each zone.

e) Create edges between all connecting roads or zones (intersection) from RNDF.

The process of creating edges for vertices within a road, within a zone or in between roads and zones is each different. The following three subsections give a brief background of each these three areas and explain how edges are created for each area.

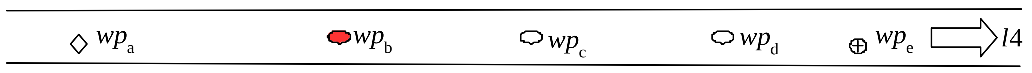
6.1.1 Edges within Road

A road, *r* is a collection of neighboring lanes flowing in the same direction with no physical median dividing them. Set of roads, $R = \{r_1, r_2, .. r_k\}$, where each road $r = \{l_1, l_2 .. l_n | \forall l_i, 0 < i < n, (l_i.direction() = l_{i+1}.direction()) \wedge (navigable(l_i, l_{i+1}) = True)\}$. The lanes l_i are number in the order of their occurrence on the segment (left to right or right to left). Here, the function *navigable* $(l_i, l_{i+1}) = True$, implies that there is no physical median dividing the neighboring lanes l_i and l_{i+1} and hence a lane change is permissible between the lanes. A road captures the idea of a collection of lanes which one can use to drive from one point to another without going into specifics of which lane to drive on.

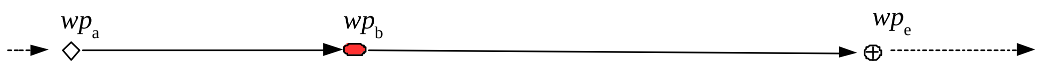
Creating edges for vertices on a single-lane road is straight-forward, while creating edges for a multilane road requires some extra work. Note that the definition of a road in the rndf_graph representation differs from the commonly used definition of a road. In common use, a road includes the set of lanes flowing in opposite directions as in a 2-way road, while in the rndf_graph notation these set of lanes are represented as two separate roads.

Here I describe how a road with one lane is represented in an rndf_graph, followed by how a road with multiple lanes is represented.

Single lane road: For a single lane road, all the vertices on that road are order based on their occurrence on the road. The consecutive vertices in this list are connected through an edge. The resulting set of edges E_l created for a single lane road l can be defined as follows. Here an edge (wp_i, wp_j) is included in E_l if both wp_i and wp_i belong to lane l and they occur as consecutive waypoints on the lane, as tested by the function *consecutive_lane_wps* (wp_i, wp_j).



(a) Single lane road with an entry point, exit point, checkpoint, and two non-critical points.



(b) Graph representing the single lane road. Non-critical waypoints (*NCWP*) are removed when creating the rndf_graph.

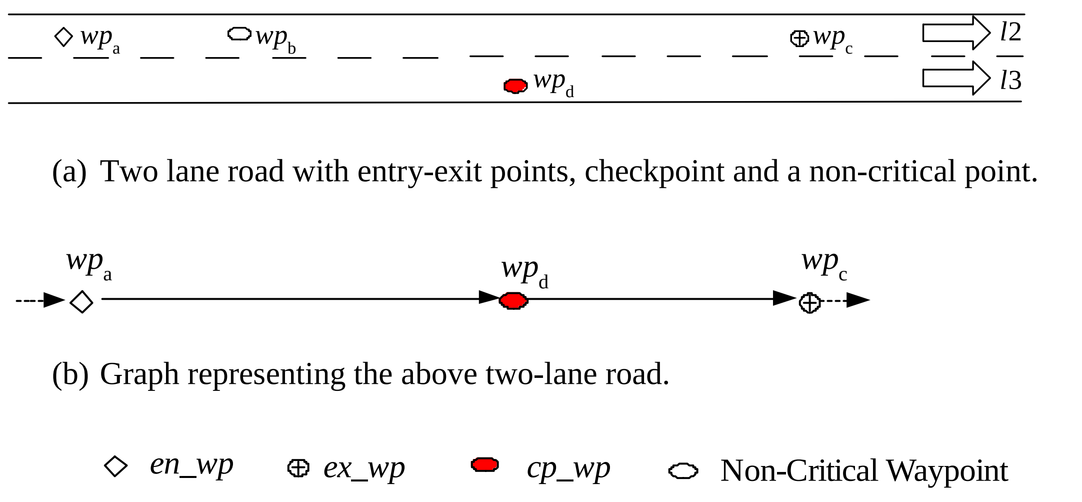
◇ *en_wp* ⊕ *ex_wp* 🟥 *cp_wp* ⬭ Non-Critical Waypoint

Figure 23: Rndf_graph representation of a single-lane road

$$(wp_i, wp_j) \in E_l \quad iff \quad wp_i \in Waypoints(l) \wedge wp_j \in Waypoints\ (l) \wedge$$
$$consecutive\_lane\_wps\ (wp_i, wp_j)$$

The graph edges are annotated with estimated time of travel between the vertices. The distance between the two vertices is computed along the intermediate Non-Critical WayPoints (*NCWP*s). The distance between the vertices, the speed limits on the lane and any extra penalty, if applied are used to annotate the edges with estimated time for travel. Figure 23 shows the graph representation for a single lane road.

Multi-lane road: Creating the edges for a multi-lane road requires merging the lanes into one unit. Lanes are merged by selecting all the *CWP*s from all the lanes and sorting them based on their occurrence along the road. The consecutive navigable vertices are connected via edges that are annotated with the estimated time of travel. Figure 24 shows the rndf_graph representation for a simple multi-lane situation with no non-navigable vertices pair.



(a) Two lane road with entry-exit points, checkpoint and a non-critical point.

(b) Graph representing the above two-lane road.

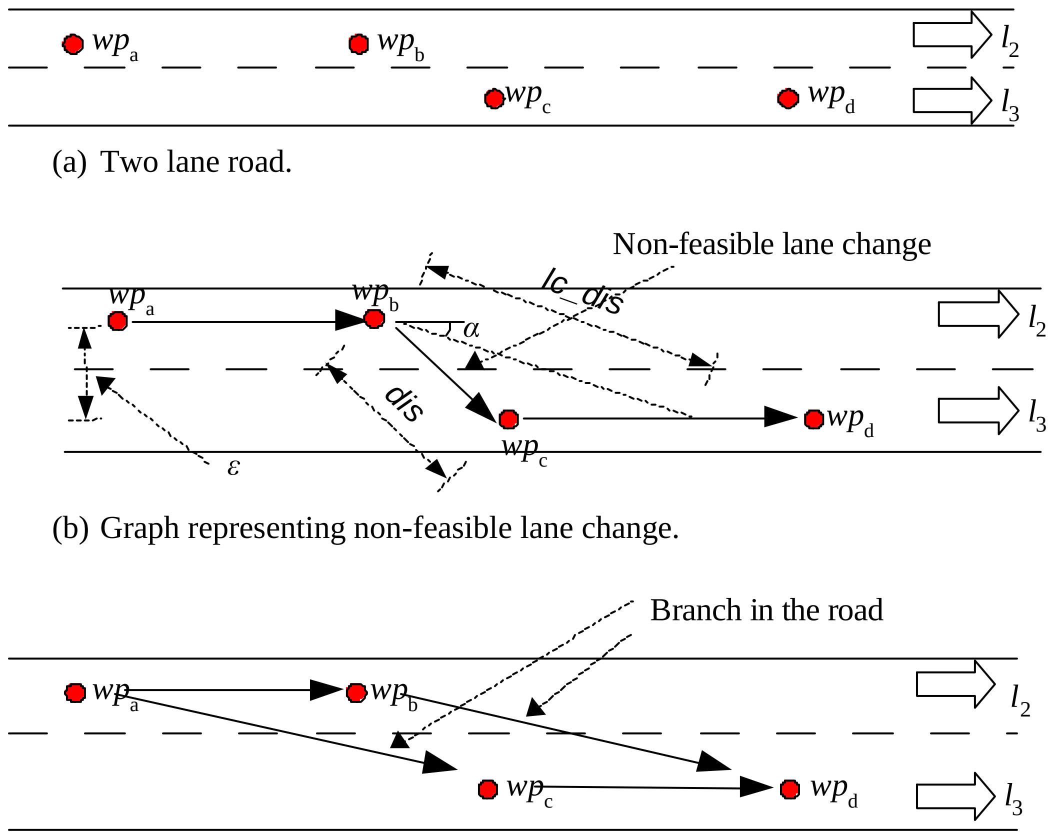
Figure 24: Rndf_graph representation of a multi-lane road

A pair of vertices (wp_i,wp_{i+1}) is not navigable if the waypoints wp_i and wp_{i+1} fall on different lanes (l_i, l_j) and the waypoints wp_i and wp_j are too close for the vehicle to switch lanes between them. An example pair of non-navigable pair of vertices is (wp_b, wp_c) as shown in Figure 25 b. That is, if the distance (*dis*) between wp_i and wp_j is smaller than required for the *lane change distance* (*lc_dis*), it is considered as not navigable. Here *lc_dis* is the suggested distance for changing lanes. The distance *lc_dis* is approximated as *ε / sin (α)*, where '*ε*' is the distance between the two lanes l_i, and l_j, and α is a safe angle for lane change. Here,

$$\varepsilon = \frac{l_1.width}{2} + \frac{l_2.wdith}{2}$$

$$lc\_dis = \frac{\varepsilon}{\sin(\alpha)}$$

$$dis = \sqrt{(wp_{i+1}.y - wp_i.y)^2 + (wp_{i+1}.x - wp_i.x)^2}$$



(a) Two lane road.

(b) Graph representing non-feasible lane change.

(c) Branch in the graph representing feasible lane change.

Figure 25: Multi-lane road merging using branching

Non-navigable vertices are handled by adding a branch around these vertices, as shown with a simple example in Figure 25. In this example all the *wp*s are assumed to be *CWP*s and are included in the rndf_graph representation. Here the waypoints wp_b and wp_c are too close to be navigable. Hence, no edge is added from vertex wp_b to wp_c in the rndf_graph representation. Instead two new edges e_{ac} and e_{bd} between wp_a to wp_c and wp_b to wp_d respectively are added. The edge e_{ab} followed by e_{bd} represents the path for driving on lane l_2 from wp_a to wp_b and then changing to lane l_3 to reach $wp_{d,}$ or changing from lane l_2 at wp_a to lane l_3 at wp_c and then continue on l_3 to reach wp_d. These extra edges represent the alternative *navigable* paths (*wp* sequence) that can be included in the MDF mission.

```
// Input: WP = {$wp_1$, $wp_2$… $wp_n$}
// Output: set of rndf_edges E
// Here, N = |$CWP$| ,
// The function nav ($wp_i$, $wp_j$) returns true if the path from $wp_i$ to $wp_j$ is navigable.
for i=0 to n-1
begin
        for j=i+1 to n
        begin
               if (nav ($wp_i$, $wp_j$))
                       Add edge from $wp_i$ to $wp_j$ to E
                           break
              endif
        end;
        for k=j+1 to n
        begin
             if (nav ($wp_j$, $wp_k$))
                    break
           endif
           if (nav ($wp_i$, $wp_k$))
                     Add edge from $wp_i$ to $wp_k$ to E
             endif
      end
end
```

Figure 26: Pseudo code for creating rndf_graph edges for a multi-lane road

Merging multiple lanes can get complex with an increase in the number of lanes and the combination of close waypoint. Following is the set representation of the all edges that should be added in this process of merging lanes. Here, *nav* (wp_i, wp_j) is true if path from wp_i to wp_j is navigable. Figure 26 gives the pseudo code for an optimized method for forming these edges.

$$E_{road} = E_{dir} \cup E_{Idir}$$

$$\left(wp_i, wp_j\right) \in E_{dir} \quad iff \quad 1 \le i < |CWP|, i < j \le |CWP|, nav\left(wp_i, wp_j\right) \wedge$$

$$\forall k, i < k < j, \neg nav\left(wp_i, wp_j\right)$$

$$(wp_i, wp_m) \in E_{Idir} \quad iff \quad 1 \le i < |CWP|, i < m \le |CWP|, \left(wp_i, wp_j\right) \in E_{Idir} \ \wedge$$
$$nav(wp_i, wp_m) \wedge \ \exists wp_j, i < j < m, \left(wp_i, wp_j\right) \in E_{dir} \wedge$$
$$\forall k, j < k \le m, \neg nav\left(wp_j, wp_k\right)$$

6.1.2 Edges within Zone

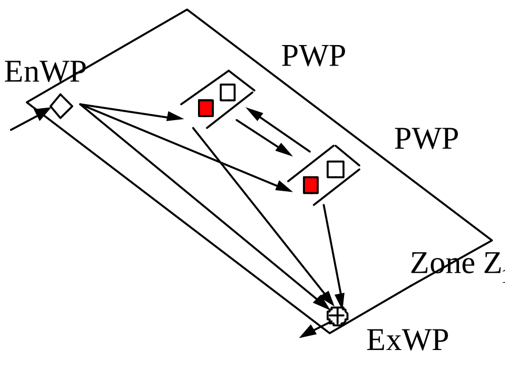
A zone is a free-travel area with optional parking spots. Zones have three types of waypoints that are of interest. They are: entry points, *EN_WP*s to enter the zone, exit points, *EX_WP*s to leave the zone, and parking waypoints, *PK_WP*s at the parking spots. These three types of waypoint vertices require four types of edges, as listed below (refer Figure 27).

a) *EN_WP* -> *EX_WP*: Edges from each entry point of the zone to each exit point of the zone. These edges represent the possible action of just passing through the zone or to reach an *EX_WP* or *EN_WP* that is marked as a checkpoint.

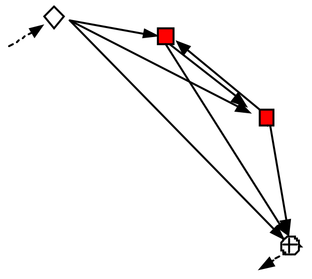
b) *EN_WP* -> *PK_WP*: Edges from each entry point to each parking spot. These edges represent the action of entering the zone to reach a parking checkpoint.

c) *PK_WP* -> *PK_WP*: Edges from each parking spot to each other parking spot. These edges represent the action of reaching from one parking point to another parking point.

d) *PK_WP* -> *EX_WP*: Edges from each parking spot to each exit point. These edges represent the action of leaving the zone through any of the exit points from a parking spot.



(a) Zone with two checkpoints.



(b) Graph representation of the zone.

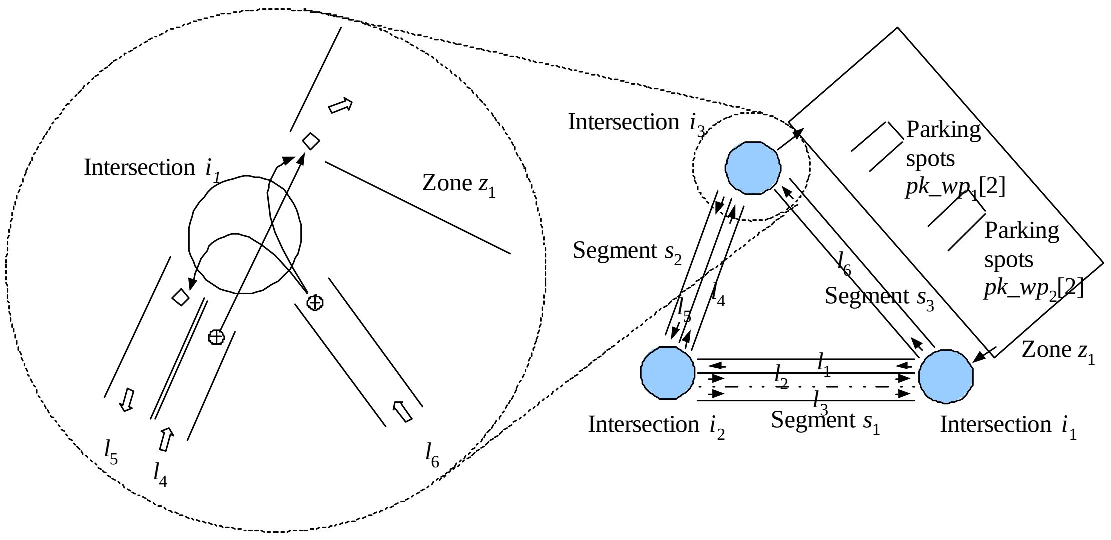
Figure 27: Rndf_graph representation of a zone

6.1.3 Edges for Intersection

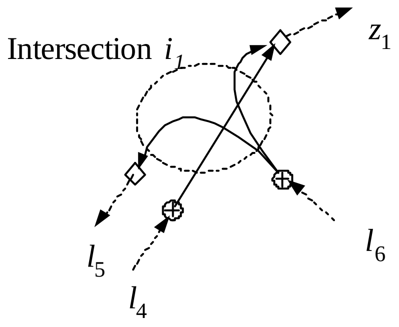
An intersection is a region where multiple lanes or zones are connected. Creating rndf_graph edges E_{int} for intersection area is straightforward. For every vertex in the rndf_graph that represents an exit waypoint of a lane or a zone, add edges to all of the vertices representing its allowed entry waypoint pairs as specified in the RNDF.

The resulting set of edges created for intersection region E_{int} can be defined as follows. Here an edge (wp_{i}, wp_{j}) is included in E_{int} if wp_{i} is an exit waypoint, wp_{j} is an entry waypoint and wp_{j} is one of the entry pair of wp_{i}. Here, *entry_pair* (*wp*) returns the set of entry waypoints which can be reached from *wp*.

$$(wp_i, wp_j) \in E_{int} \quad iff \quad wp_i \in Ex\_WP \ \wedge wp_j \in En\_WP \ \wedge wp_j \in entry\_pair(wp_i)$$



(a) Intersection i_1 joining two lanes and a zone.



(b) Rndf_graph representing the intersection. For visual clarity only the edges originating from lane l_6 are displayed.

Figure 28: Rndf_graph representation of an intersection

Figure 28 (a) shows an example intersection area where two segments and a zone are connected. Figure 28 (b) shows the rndf_graph representation of this intersection. Here the *ex_wp* from l_6 is connected to *en_wp* on lane l_5 and zone z_1. Similarly *ex_wp* from lane l_4 is connected to *en_wp* on z_1.

6.2 Mission Planner

The Mission Planner module searches the rndf_graph created in the previous step to find a sequence of connected vertices that represents the path to reach all the checkpoints. This sequence of waypoints is known as Mission Plan and provides a plan indicating where to enter which road (or zone) and where to exit a road, or a zone, in order to reach all the checkpoints *CP*.

The process of finding the Mission Plan starts by first mapping present position of the AGV onto the rndf_graph to find W_{start}, a waypoint preceding AGV in the rndf_graph. This waypoint is added as the preceding element cp_0 to the input MDF mission = [cp_1, cp_2, .. cp_n] to get new sequence CP=[cp_0, cp_1, cp_2… cp_n].

The rndf_graph is searched for a path between consecutive cp_{i-1}, cp_i using *GraphSearch* (cp_i, cp_j). The resulting sequence of waypoints is filtered to leave only the EN_WP, EX_WP and the goal waypoint cp_j. This filtered sequence of waypoints is called *sub-goal waypoints* (swp_i) which does not include the start node cp_i. This process is repeated to determine swp_1, swp_2, .. swp_n corresponding to each checkpoint from MDF mission. These sub-goal waypoints are appended to w_{start} to form a single sequence of waypoints known as Mission Plan. Following is the formal representation of these steps:

$Mission\ Plan = missionPath(w_{start}\,, CP)$

$missionPath\ (cp_0, [cp_1, cp_2\ ...\ cp_n]) = [cp_0] + swp_1 + swp_2 + \cdots swp_n$

Where, $swp_i\ = ExEnPath(cp_{i-1}, cp_i)$ and '+' is concatenation of sequences

$swp_i\ = Filter\big(GraphSearch(cp_{i-1}, cp_i)\big)$

Where, $GraphSearch\ \left(cp_i, cp_j\right) = \left[wp_1, wp_2, ..wp_m, cp_j\right]$, A quickest path from cp_i to cp_j determined by searching the time-annotated rndf_graph. This sequence of waypoints does not include the start waypoint cp_i, but includes the goal waypoint cp_j.

And the function $filter(S)$is defined as follows:

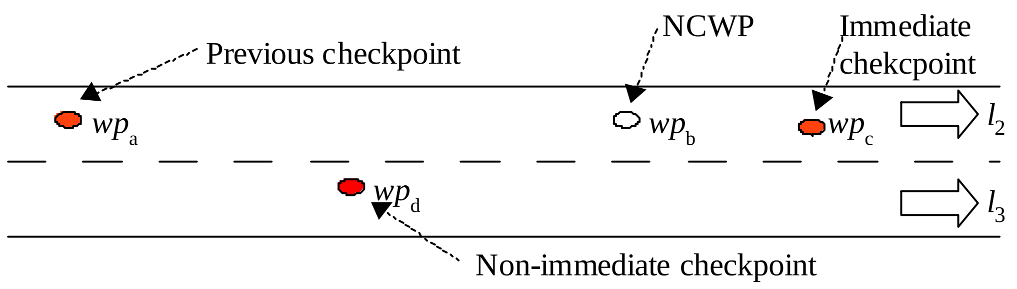
$Filter([]) = []$,

$Filter([x]) = [x]$,

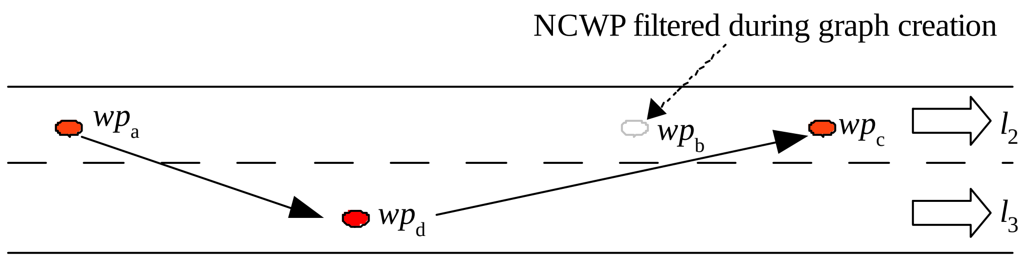
$$Filter([x].S) = \begin{cases} [x] + Filter(S) & if\ x\ \in EX\_WP \\ [x] + Filter(S) & if\ x \in EN\_WP \\ Filter(S) & else \end{cases}$$

The filtering is required in the above process as the rndf_graph include the vertices representing all the checkpoints included in the MDF mission. While searching the graph for checkpoint cp_i, all the other checkpoints that are included in the graph vertices are of no importance. Hence all these checkpoints are filtered out. Here the checkpoint cp_i being searched for is called immediate-checkpoint.

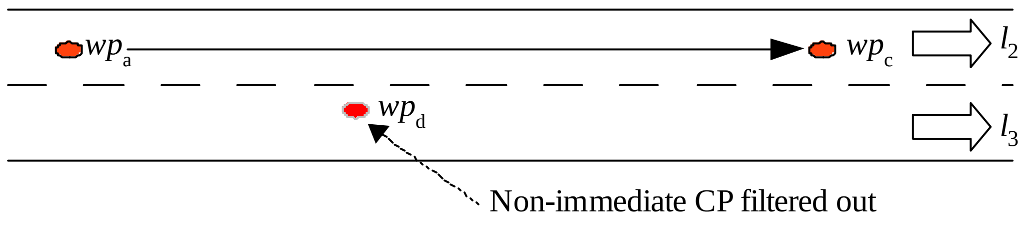
Figure 29 shows an example scenario to illustrate the above filtering process. Here, consider a MDF mission to reach wp_a, wp_c, and wp_d in that order. Since they are part of the MDF mission, all three *wp*s are included in the rndf_graph representation of the road. However, when searching for a sub-mission to reach wp_c from wp_a, wp_d as a checkpoint has no role. Instead, wp_d leads to an unnecessary lane changing from l_2 to l_3 and back to l_2. Hence, non-immediate checkpoints like wp_d are filtered out.



(a) Two lanes road as represented in RNDF definition, with wp_a, wp_c, wp_d as checkpoints from the MDF.



(b) rndf_graph representation of the road network, with non-critical waypoint wp_b, removed.



(c) Mission plan to reach wp_c, from wp_a: Checkpoint wp_d filtered from the plan as it is not the immediate checkpoint

● Critical waypoint ○ Non-critical waypoint

Figure 29: Filtering graph search waypoint list to get the Mission Plan

6.3 BP Extractor

The Mission Plan planned in the previous step is used to find a sequence, σ of BPs that can accomplish the MDF mission. Consecutive waypoints (wp_i, wp_{i+1}) in the Mission Plan are logically connected and represent a specific task. These tasks can be achieved by a sequence $\sigma_{wpi,\ wpi+1}$ of BPs. As an example, let waypoints pair (wp_1, wp_2) be a consecutive waypoints in a Mission Plan belonging to two adjacent lanes (l_1, l_2) of a segment. To drive between these two *wp*s, system needs to: a) drive on lane $l_{1,}$ for a certain distance, b) change to lane l_2, and c) continue to drive on lane l_2 to reach wp_2. This is achieved through a sequence of three Basic Planners (BPs): $\sigma_{wp1,wp2} = [fl\_bp_1, cl\_bp_1, fl\_bp_2]$, where fl_bp_1 drives on lane l_1 for certain distance, cl_bp_1 changes to lane l_2 and fl_bp_2 drives on l_2 to reach wp_2.

The sequence $\sigma_{wpi,\ wpi+1}$ of BPs that can drive between a consecutive pair of waypoints in the mission plan is determined independently and then concatenated to get a single sequence σ of BPs that will accomplish the whole mission. Determining the sequence of BPs that will drive between a pair of waypoints can be achieved through a logic-based planning approach or decision-rules based planning approach.

In the Logic-based approach as explained in Section 6.3.1, the present planning problem can be represented in a logic-based language such as an ADL and use a standard logic-based planner to search for a sequence σ of BPs that can accomplish the mission. This approach helps to achieve easy plug-ability of new BPs, as explained in section 6.3.2.

The decision-rule based planner encodes the predetermined decisions of what sequence of BPs can be used for each possible combination of waypoint pairs. This approach was implemented and tested while preparing for Urban Challenge and is explained in Section 6.3.3.

6.3.1 Logic-based Planning for BP Extractor

Here I show how the Basic Planner (BP) Extraction problem statement can be represented using one of the logic-based representation language, namely Action Description Language

(ADL). In this language, problem constraints and mission are represented as propositional literal states or first-order states, and BPs are represented as action. This language representation is used with a standard planning approach to determine a sequence of actions (BPs) that can achieve the goal state (mission). The planning algorithms that can be used for such a representation includes forward state-space search, backward state-space search (Russelll and Norvig 2003), partial-order planning (Nilsson 1998), and GraphPlan (Russell and Norvig, Planning Graph 2003).

The ADL notation for symbols is different from the set theory notation used in the rest of the document. In ADL notation, a capitol letter symbol such as 'L' represents a constant, while in the set theory it represents a set of values. Similarly a small letter symbol such as 'l' represents variable in ADL representation, while in the set theory it represents a specific value. I use ADL notation for this subsection and hence different from the rest of the document.

In the rest of this section I represent a simple case of BP Extraction problem in ADL representation. I then use forward-state space search algorithm to plan a BP sequence, σ for a sample waypoint pair.

<u>ADL representation</u>

Table 8 outlines the ADL language which includes a list of propositional literals, list of first-order literals states, and a list of actions. This terminology will be used to represent a sample problem (*Prob-1*) shown in Figure 30. Here the goal is to reach waypoint wp_2 from waypoint wp_1.

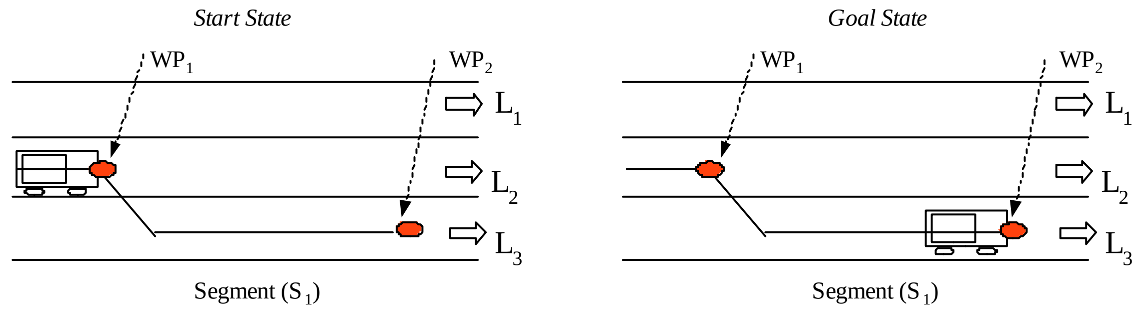


Figure 30: Graphical representation of start and goal state of Prob-1

Table 8: ADL language representation

Symbol	*Represents*
	Propositional literals
Left	Represents the left side of a lane
Right	Represents the right side of a lane
	First-order literals
Right_Lane (l_1, l_2)	Represents that l_2 is the immediate right neighbor of l_1
Left_Lane (l_1, l_2)	Represents that l_2 is the immediate left neighbor of l_1
WPLane (wp)	Represent the lane to which *wp* belongs
	First-order literals (states)
On_lane (*l*)	Represent the state of being is on lane *l*
On_segment (*s*)	Represent the state of being is on segment *s*
Pass_WP (*wp*)	Represent the state of passing waypoint *wp*
	Actions
CL_BP (*Left*)	Action to switch to a left lane
CL_BP (*Right*)	Action to switch to a right lane
FL_BP (*wp*)	Action of following the lane till waypoint *wp*

The ADL representation of this problem (*Prob-1*) is shown in Figure 31. The representation has three sections, namely, *Init*, *Goal*, and *Action*s. Here, the *Init* section describes the initial state of the problem. *Goal* section represents the required goal state the

$Init\ (On\_Lane(L_2) \land On\_Segmnet() \land Pass\_WP(WP_1))$

$Goal\ \left(Pass\_WP(WP_2)\right)$

$Action(CL\_BP(LEFT)),$
$\quad PRECOND: On\_Lane(l_1) \land Left\_Lane\ (l_1, l_2)$
$\quad EFFECT: On\_Lane(l_2) \land\ \neg\ On\_lane(l_1))$

$Action(CL\_BP(RIGHT),$
$\quad PRECOND: On\_Lane(l_1) \land Right\_Lane\ (l_1, l_2)$
$\quad EFFECT: On\_Lane(l_2) \land \neg\ On\_Lane(l_1)$

$Action(FL\_BP(wp),$
$\quad PRECOND: On\_Lane(l) \land (WPLane(wp) = l) \land \neg\ Pass\_WP(wp)$
$\quad EFFECT: Pass\_WP(wp))$

Figure 31: ADL description of a sample BP Extraction problem: *Prob-1*

system should reach. And *Actions* section lists the actions that are allowed in the problem. Each action description has three parts, namely, syntax, precondition and effects. The syntax part describes the name and input arguments of the action. Precondition part lists the conditions required to perform the action and the effects part list the changes that are brought about by the action.

Forward-state space search algorithm

The present problem (*Prob-1*) can be solved using logic-based planning algorithms such as partial-order planning, forward state-space search and backward state-space search algorithm. As an example in this section I use forward state-space search algorithm on *Prob-1*, as shown in Figure 32. The search starts with a start node representing the initial conditions, as mentioned in the problem description in Figure 31, and explores the new states that can be reached by taking each valid action. An action is considered valid from a state, if all the preconditions of the action are satisfied by the state. This exploration of new states is performed in a bread-first order.

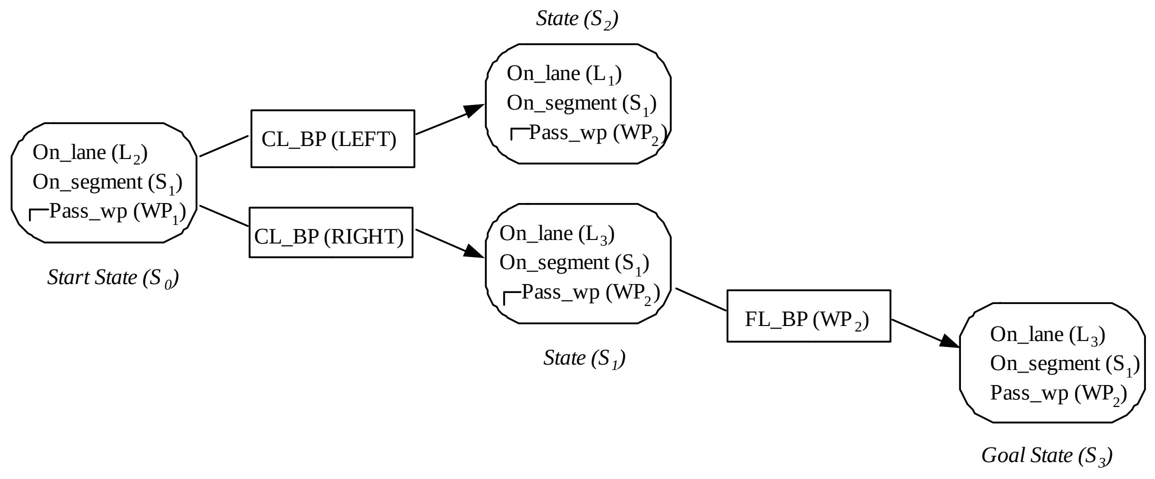


Figure 32: 'Forward state-space' search algorithm on *Prob-1*

From the start state S$_0$ described as $OnLane\ (L_2) \wedge OnSegment\ (S_1) \wedge Pass_{WP}\,(WP_1)$, there are two valid CL_BP actions {CL_BP (LEFT), CL_BP (RIGHT)} and one invalid FL_B action {FL_BP (wp)}. The FL_BP (wp) is invalid as it could not satisfy the preconditions to take any of WP$_1$ and WP$_2$ as an action argument. To be more specific, the precondition of $\neg\ Pass\_WP(WP_1)$ is not satisfied for taking WP$_1$ as an argument, and the precondition of $(L_1 = L_2)$ is not satisfied for taking WP$_2$ as an argument.

Actions CL_BP (RIGHT) and CL_BP (LEFT) lead to two new states S$_1$ and S$_2$ respectively. The preconditions for action FL_BP with an argument WP$_2$ are now satisfied from states S$_1$. This action leads to a new state (S$_3$) which satisfies all the goal conditions, thus successfully completing the search.

6.3.2 Plug-ability in BP Extractor

In a logic-based planning approach for BP Extractor, each Basic Planner (BP) type is represented as an action with a list of preconditions and a list of effects the BP is expected to make. This language is used by the planning algorithm to determine the sequence σ of BPs required for navigating between a pair of waypoints. Adding a new BP to the CB_PP requires adding a new action and, if required, a few state symbols to the language. The planning algorithm which determines the sequence of actions (BPs) is not tied to any of these actions. Hence adding a new action to the language should not affect the planning algorithm, and thus allowing easy addition of new BPs (actions) to the BP Extractor.

6.3.3 Decision-rules Based BP Extractor

This approach encodes the predetermined decisions of what sequence of Basic Planners (BPs) to use for each possible combination of waypoint pairs. These decisions are represented as decision-rules as described in this section. For simplicity, the rules can be classified into three categories depending on where the waypoints wp_1, wp_2 are located on the RNDF. The categories include wps that are:

(i) within the same zone

(ii) on different segment or zones

(iii) on same segment.

6.3.3.1 Within the same zone

Waypoints within a zone can either be a perimeter point, to enter or leave the zone, or can be a parking point, specifying a parking spot. These two waypoint types will results in four possible combinations for waypoint pairs $\{wp_1, wp_2\}$. In this section, wp_1 is represented as wp_{prev} and wp_2 is represented as wp_{next}. The sequence σ of BPs that can drive between these different waypoint pairs is listed in Table 9.

Table 9: BP Extraction rules for within zone WPs

WP_{prev}	WP_{next}	*Sequence of Basic Planners* $(\sigma_{wp_{perv}\ wp_{next}})$
Zone_perimeter	Zone_perimeter	[*zn_bp* (wp_{next})]
Zone_perimeter	Parking_spot	[*zn_bp* (wp_{next}), *pk_bp* (wp_{next})]
Parking_spot	Zone_perimeter	[*up_bp* (wp_{next}), *zn_bp* (wp_{next})]
Parking_spot	Parking_spot	[*up_bp* (wp_{next}), *zn_bp* (wp_{next}), *pk_bp* (wp_{next})]

Zone-Navigator Basic Planner, ZN_BP as explained earlier plans a path to drive to a specific position at a given orientation. ZN_BP takes this destination position and orientation as its goal. ZN_BP's destination position is equal to the wp_{next} position, while the destination orientation depends on the wp_{next} type. If the wp_{next} is a perimeter point, then the ZN_BP's destination orientation is equal to the orientation of the normal to the zone's perimeter at wp_{next}. If the wp_{next} is a parking point, then the ZN_BP's destination orientation is equal to the orientation of the parking spot.

Similarly, PK_BP and UP_BP take the destination position and direction as their local goals. These values are obvious for PK_BP, as it is equal to the parking spot's position and orientation. For UP_BP, the destination position and orientation is computed depending on the relative position of the wp_{next} from the wp_{prev}. Here, wp_{next} can be another parking spot or a zone perimeter point.

6.3.3.2 On different segments or zones

If wp_{prev} and wp_{next} are on different segments or zones, then wp_{prev} and wp_{next} should have been connected by an exit-entry connection. For such a waypoint pair, an Intersection Basic Planner *it_bp* (wp_{next}) can drive between the *wp*s. $\sigma_{wp_{pre\ v}\, wp_{next}}$ = [*it_bp*(wp_{next})]

6.3.3.3 On same segment

This section list the decision rules required for determining the sequence $\sigma_{wp_{prev}\ wp_{next}}$ of BPs that can navigate between the waypoint pair (wp_{prev}, wp_{next}) belonging to the same segment. These decision rules can be further be classified into two categories based on if the waypoints belong to the same lane or two different lanes within the segment.

a) Waypoints belong to same lane (of the same segment):

Following is the decision rule when wp_{prev} and wp_{next} belong to the same lane. As the rule describes though the *wp*s belong to the same lane, there could be a stop sign at wp_{prev} requiring using an IT_BP. Another interesting case is that the lane that the *wp*s belong to might have a loop and waypoint pair (wp_{prev}, wp_{next}) might be jumping ahead on the lane. An example situation is shown in Figure 33. Here the waypoint pair (wp_1, wp_4) belong to the same group but require an IT_BP to navigate from the wp_1 to wp_4.

In the following decision rule *consecutive_lane_wps* (wp_{i}, wp_{j}) returns true if wp_{j} occur immediately after wp_{i} on the lane.

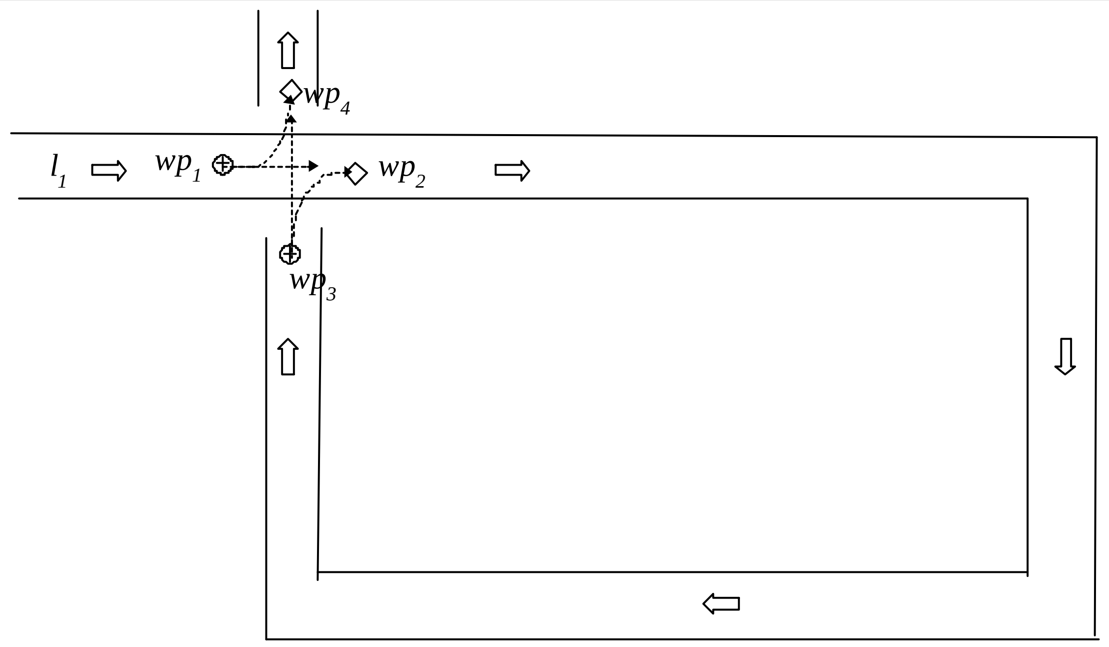


Figure 33: Single lane looping and interacting with itself

$$\sigma_{\mathrm{wp\ prev}\quad \mathrm{wp\ next}} = \begin{cases} [it\_bp\ (wp_{next})] & if \quad wp_{prev} \in Ex\_WP \wedge stop\_sign(wp_{prev}) \\ [it\_bp(wp_{next})] & if \quad wp_{prev} \in Ex\_WP \wedge \neg consecutive\_lane\_wps(wp_{prev}, wp_{next}) \\ [fl\_bp(wp_{next})] & else \end{cases}$$

b) Both the waypoints belong to two different lanes (of the same segment):

Following is the decision rule when wp_{prev} and wp_{next} belong to two different lanes of the same segment. If these lanes are oriented in the same direction then a CL_BP should switch lane followed by a FL_BP to reach wp_{next}. If the lanes are in the opposite direction, connected as an exit-entry pair with no other lanes or zones connected to this exit waypoint wp_{prev}, then the segment is a dead end and requires a U-turn. In the following rule, *entry_pair* (*wp*) returns a set of entry pairs for wp_{prev}. If this set equals $\{wp_{\mathrm{next}}\}$, then there exist no other lanes or zones connected to this exit waypoint wp_{prev}.

$$\sigma_{wp_{prev}} \ \mathrm{wp}_{\mathrm{next}}$$

$$= \begin{cases} [cl\_bp(lane(wp_{next}), fl\_bp(wp_{next})] & if \quad lane(wp_{prev}).direction = lane(wp_{next}).direction \\ [ut\_bp(lane(wp_{next}))] & if \quad wp_{prev} \in Ex\_WP \wedge wp_{next} \in En\_WP \wedge \\ & \quad entry\_pairs(wp_{prev}) = \{wp_{next}\} \\ [it\_bp(wp_{next})] & else \end{cases}$$

7 Supervisor

The Supervisor module uses the sequence (σ) of Basic Planners (BPs) determined by the HLP to plan a steering path for the AGV to follow. The steering path is used by the Steering Controller module (refer Figure 1), which is not part of Path Planner, to generate the low level steering, gas, and brake commands(Herpin, et al. 2007). The Supervisor module performs this path planning cycle every 50 milliseconds publishing a steering path at 20Hz. In each path planning cycle, the Supervisor module performs the following five tasks (refer Figure 34).

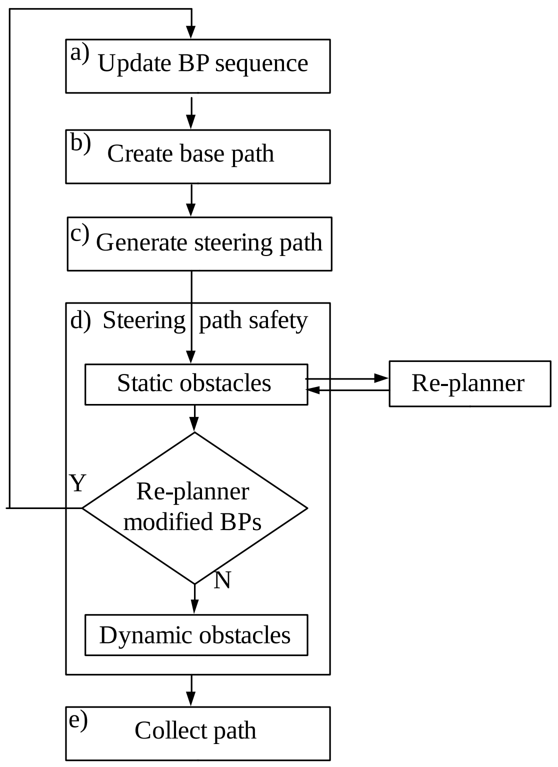


Figure 34: Control flow of Supervisor module

a) Update the BP sequence by removing the BPs that have completed their tasks.
b) Create a dummy base path for the first BP in the BP sequence.
c) Trigger the BPs in the BP sequence to plan their path.
d) Verify that each of BP’s path segments is safe to follow with respect to static obstacles. If not safe, inform the Re-Planner. If the Re-Planner modified the BP sequence, then the Supervisor module restarts from the first step. Else, the Supervisor module continues to check if the path segments are safe to follow with respect to dynamic obstacles.

```
//Function Name: supervisor_main_loop ($\sigma$)
//Input: $\sigma$
//Output: $P_{output}$, path for an AGV to follow
Label: start
    $\sigma$ = update_sequence ($\sigma$)                  // Update $\sigma$ by removing completed bps
    if (empty($\sigma$))                                   // If done with all bps in $\sigma$ initiate
    then                                                   //       termination of CB_PP
        initiate termination of CB_PP
        return
    endif
    $P_{base\_input}$ = create_base_path ($P_{prev}$, $\sigma[0]$ ) // Generate base input path for 1st bp
    plan_steering_path ($\sigma$, $P_{base\_input}$)       // Trigger bps to generate their path
    // Check for safety against static obstacles, here obstacle_speed should be zero
    (bp, $i_{path}$, obstacle_speed) = on_path_obstacles ($\sigma$, STATIC)
    if (bp != NULL)                                        // If found bp with an obstacle on its path at
    then                                                   //       $i_{path}$ index of its path.
         set_safety_speed_limits ($\sigma$, bp, $i_{path}$,0)
    endif

    $\sigma_{new}$ = re_planner ($\sigma$, bp, $i_{path}$)  // Check with Re-planner for any updates
    if ($\sigma_{new}$ != $\sigma$)                        // If modified $\sigma_{new}$ then start the beginning
    then
        $\sigma = \sigma_{new}$
        goto start
    endif
    handle_dynamic_obstacles ($\sigma$)                    // Detect and handle dynamic obstacles
    $P_{output}$ = collect_path ($\sigma$, $P_{base\_input}$)  // Collect the path for publishing
    $P_{prev}$ = $P_{output}$                              // Update $P_{prev}$ path
    publish ($P_{output}$)                                 // Publish the path for AGV to follow
```

Figure 35: Pseudo code illustrating the steps followed by Supervisor module: *supervisor_loop*()

e) Collect the paths generated by the each BP and publish the combined steering path.

Figure 35 gives the pseudo code of the *supervisor_main_loop* (), which implements the above mentioned steps. Here, *update_sequence* (σ) function removes the top of the sequence BPs that have completed performing their capability. This step is further explained in Section 7.1. If all the BPs from the σ sequence are done, the Supervisor initiates termination of CB_PP. Else, the Supervisor continues to generate the base path, P_{base_input} by calling *create_base_path* (P_{prev}, $\sigma[0]$) function. This function is further explained in Section 7.2. The generated base path is used to call *plan_steering_path* (σ, $P_{\text{base\_input}}$) function explained in Section 7.3 to walk through BPs in the sequence and trigger the BPs to generate their path. This generated path is tested for safety against static obstacles using function on_path*_path_obstacles* (σ, STATIC). Upon detecting a static obstacle on the path, the speed limits on the path are set to maintain minimum safe distance from the detected static obstacle. This is performed by *set_safety_speed_limit* (σ, *bp*, i_{path}) function. These two steps are explained in more detail in Section 7.4. The *re-planner* is then called to let it perform any updates to BP sequence to give a new sequence σ_{new} . If σ_{new} is different from σ, the σ is updated to σ_{new} and the Supervisor modules starts from the first step by jumping to label *start*. Else the Supervisor module continues to handle any dynamic obstacles that can pose a danger. This is performed by the *handle_dynamic_obstacles* () function. Finally, *collect_path* () function explained in Section 7.6, walks through the BP sequence to collect a single final path for the AGV to follow. The final step of *publish_path* communicates the results to the Steering Controller module. For CB_PP implementation, *publish_path* () function communicates its path to the Steering Controller module by pushing the final path onto a queue implemented in a shared memory (Venkitakrishnan 2006).

7.1 Update the Basic Planner Sequence

In this step, the Basic Planners (BPs) which has completed performing their capabilities are removed from the top BP sequence. The decision of whether a capability is complete is left for the BPs to decide. Though the present set of all BPs decide on completing their capabilities when their planned path P_{BP} is covered by the AGV, there is no hard definition on this decision.

Figure 36 shows the pseudo code of this step. The Supervisor module at this step, as shown in Figure 36, queries the first BP in the sequence to check if it has completed its capability. If so, the BP is removed and the next BP in the sequence is queried for completion. This process is repeated until the first BP in the sequence that has not completed its capability or until there are no BPs in the sequence. If there are no more BPs left in the sequence, the Supervisor module initiates termination of the Path Planner.

```
//Function Name: update_sequence (σ)
//Input: σ
//Output: updated σ

while (not_empty (σ) && !σ[0].is_completed ())
begin
    remove_first_element (σ)
end
```

Figure 36: Pseudo code for updating the BP sequence, σ: *update_sequence* ()

By letting the BPs make the decision about completing the capability, the Supervisor module is able to operate without knowing the specifics of the BPs. This allows the Supervisor module to handle any new BP added to the Path Planner and thus achieve easy plug-ability of BPs. The Supervisor module uses the BP's common interface *is_completed* () function specified in Figure 8 to make this query.

7.2 Create Base Path

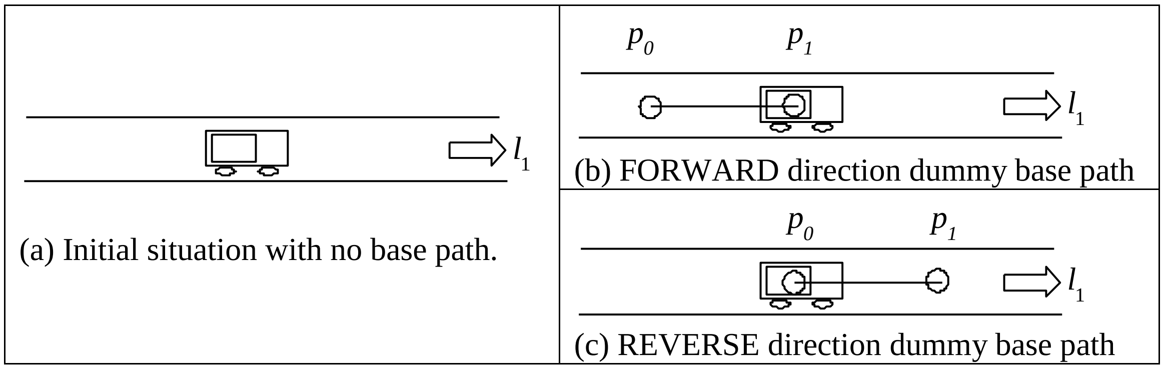
The updated Basic Planner (BP) sequence contains a sequence of BPs that are yet to accomplish their capabilities by generating the paths for the AGV to follow. To follow the

Basic-Path protocol BPs in the sequence needs an input base path $P_{\text{base\_path}}$. The present step generates this input base path for the first BP in the sequence.

Figure 37 provides the pseudo code for generating the base path. For the very first path planning cycle of the CB_PP, a dummy base path is created using the *dummy_base_path* () function. Else, the part of the previously generated path, P_{prev} is used as base path. If the P_{prev} path is FORWARD, the part of the path behind the AGV is used, else if the P_{prev} path is REVERSE, the part of the P_{prev} path, ahead of the AGV is used as base path. This generated base path is used to check if the first BP in the BP sequence, *bp*, is going to generate a path in the same direction. If bp path's direction is going to be different, a dummy base path is generated in the bp's path direction using *dummy_base_path* () function.

```
//Function Name: create_base_path ($P_{prev}$, bp)
//Input: $P_{prev}$, bp
//Output: $P_{base\_path}$

if ( first path planning cycle)
then
    return dummy_base_path (FORWARD)
endif

generate $P_{base\_path}$ as part of $P_{prev}$:
    behind the AGV's present position if Pprev.direction = FORWARD
    ahead of the AGV's present position if Pprev.direction = REVERSE

if (direction ($P_{base\_path}$) != bp.get_path_direction ($P_{base\_path}$))
then
    return dummy_base_path (bp.get_path_direction ($P_{base\_path}$)
endif
return $P_{base\_path}$
```

Figure 37: Pseudo code for generating base path for the first BP in the sequence



(a) Initial situation with no base path.

(b) FORWARD direction dummy base path

(c) REVERSE direction dummy base path

Figure 38: Placement of dummy base path's points

Figure 38 shows the dummy path generated by *dummy_base_path* () function for both FORWARD and REVERSE direction. The generated base path contains two points, [p_0, p_1]. For FORWARD base path direction, as shown in Figure 38 (b), p_0 is set to be at a distance d_{path} behind the AGV and p_1 is set to present position of the AGV. For REVERSE direction base path, as shown in Figure 38 (c), p_0 is set to be at a distance d_{path} ahead of the AGV and p_1 is set to the present position of the AGV.

7.3 Plan Steering Path

After updating the Basic Planner (BP) sequence, the Supervisor module iterates through the BPs in the new sequence and triggers each of them to plan their path to achieve their local goal. The input base path, $P_{\text{base\_input}}$ to *plan_steering_path* () function is provided as an input to the first BP in the BP sequence which plans a path segment, P_{BP} extending the $P_{\text{base\_input}}$ to accomplish its task. The BP stores its path segment (P_{BP}) and outputs the combined path segment $P_{\text{BP\_output}} = P_{\text{base\_input}} + P_{\text{BP}}$, as show in the pseudo code of Figure 39. The output path ($P_{\text{BP\_output}}$) from one BP forms the input base path ($P_{\text{base\_input}}$) for the next BP in the sequence. This process of iterating through the BP sequence is continued until one of the four conditions is detected.

```
//Function Name: $plan\_steering\_path$ ($\sigma$, $P_{base\_path}$)
//Input: $\sigma$, $P_{base\_path}$
//Output: $P_{output}$
// Here MAX_PATH_LENGTH
$path\_left$ = MAX_PATH_LENGHT
for $i$ = 1 to | $\sigma$ |
begin
    $bp = \sigma[i]$
    ($P_{out}$, $path\_left$) = $bp.generate\_path$ ($P_{base\_path}$, $path\_left$)
    if ($path\_left$ <= 0 || $bp.stop\_path\_generation$ ())
    then
        break;
    endif
    $P_{base\_path} = P_{out}$
done
return $P_{out}$;
```

Figure 39: Pseudo code for triggering BPs in σ to generate their paths: *plan_steering_path* ()

a) The present BP is the last BP in the sequence.

b) The cumulative length of the path planned so far is greater than the length of path that is sufficient to plan ahead of the AGV, as set to MAX_PATH_LENGTH in Pseudo code of Figure 39. For performance reasons, the path planning is limited to a certain distance ahead of the AGV's position. Also, due to the limitation of the physical sensor's viewing distance, planning beyond this distance is not of great use. During UC this distance was set to 200 meters.

c) The BP communicates to the Supervisor module to not continue the path planning process. This is performed via *stop_path_generation* () function provided by the BP's common interface class. The BP will make such a request when it cannot plan its path all the way to the goal. Or when the BP's path is not in the same direction (*forward*/*reverse*) as that of its input base path. Some BPs plan paths that should be covered in the reverse drive. As an example, the path by UP_BP to pull out of a parking spot or the path of UT_BP to back-up during a U-turn should be followed in reverse direction.

This decision to not plan a path ahead of such a point is maintained until the present path is completely covered by the AGV. At which point, the first BP in the BP sequence wants to change the direction of the path. At this instance, a new dummy input base path ($P_{\text{base\_input}}$) is fabricated in the new direction and provided as an input to the BP by the *base_path* () function, as shown in Section 4.4.

7.4 Checking for Static Obstacles

In this step the Supervisor module iterates through the BPs in the sequence, to check for static obstacles along each BP's path that are generated in the previous step. The supervisor skips the BPs that can handle obstacles along their paths. Zone related BPs are the example BPs that can handle obstacles on their path. The Supervisor module uses the common interface function *can_handle_obstacles* (), to check if a BP can handle obstacles on its path.

The process of iterating through the BP sequence to check for static obstacles along the path is continued until:

a) A static obstacle is found on the path or
b) The BP is the last in the sequence that planned a path segment. Refer to Section 7.3 for reasons on why path planning could stop in the middle of the BP sequence.

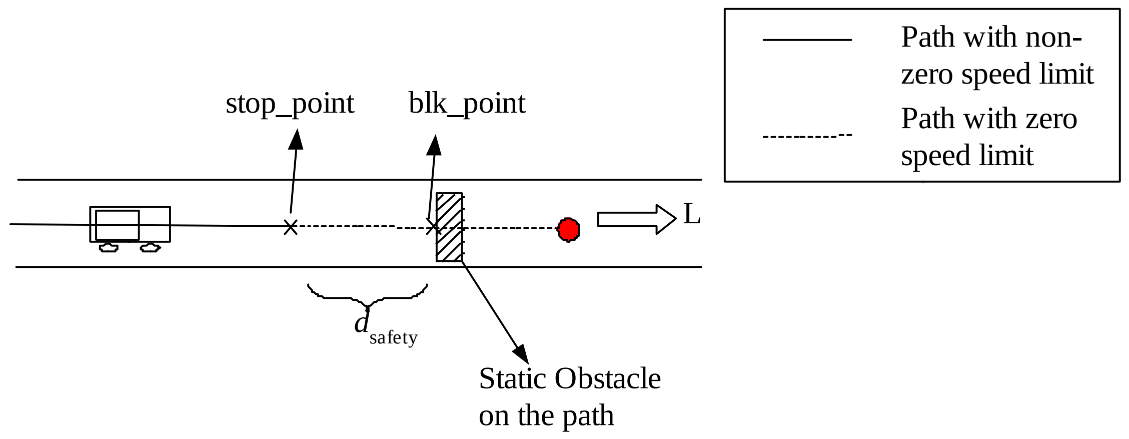
For each BP, the Supervisor module gets the planned path (P_{BP}) and the information of where on the RNDF the P_{BP} lays, lane IDs. In the present version of the architecture, the zone related BPs are expected to handle obstacles within the zone. And intersection region is considered as an extension of the lanes approaching the intersection. Hence, the area on the

```
// Function Name: on_path_obstacles ($\sigma$, motion)
//Input: $\sigma$ , motion = STATIC or DYNAMIC
//Output: (index_to_bp, index_to_bp_path, obstacle_speed)
//          return (NULL, NULL, NULL) if no obstacle is detected on the path

for $i$ = 1 to $|\sigma|$
begin
    bp = $\sigma[i]$
    if (bp.can_handle_obstacles ())
    then
        continue
    endif
    $L_{ids}$ = bp.get_lane_ids ()
    obstacles = world_state.get_obstacles_on_lanes ($L_{ids}$, motion)
    $P_{bp}$ = bp.get_path ()

    if ((path_index, obstacle_speed) = closest_obstacle_on_path ($P_{bp}$, obstacles))
    then
        return (i, path_index, obstacle_speed)
    endif
done
return (NULL, NULL, NULL)
```

Figure 40: Pseudo code for checking the obstacles on the BP's path: *on_path_obstacles* ()

RNDF that a BP can mention includes a list of lanes. Table 10 lists the lanes that would be returned for each of the road BPs and intersection BP. The Supervisor module uses this information of where the path overlays to get all the static obstacles in those regions from the World_State and checks to see if any of these obstacles fall on the path.

Figure 40 provides the pseudo code of the *on_path_obstacles* () function to detect the closest obstacle on the path. The function can check for either static or dynamic obstacles on the path, as specified by the second argument motion = {STATIC, DYNAMIC}. The function iterates through the BPs in the BP sequence and for each BP, checks if it can handle obstacles on its path. If the BP can handle the obstacle, the Supervisor skips to next BP in the sequence. Else, the Supervisor queries the BP for the list of lanes on which its path overlays. This list of lanes is used to get all the obstacles on these lanes from the World_State. These



Path with non-zero speed limit
Path with zero speed limit
stop_point
blk_point
L
d_{safety}
Static Obstacle on the path

Figure 41: Description of *stop_point* and *block_point* along the path

obstacles are mapped onto the path to detect the obstacles closest along the path. Upon detecting the obstacle on the path, it returns the index to the BP, index to BP's path with the obstacle and the speed of the obstacle. If no obstacles are detected on any BP's path, the function returns NULL for each output parameter.

Upon detecting an obstacle on the path (at Blk_point), the Supervisor module uses the *set_safety_speed_limits* () function to walk back along the path by a safety distance (d_{safety}), as shown in Figure 41, and informs the corresponding BP to set the speed limit of the path at that point to zero. This point is called stop_point. This is the closest point an AGV can drive towards the obstacle. It then informs the Re-Planner module about the blockage. The

Table 10: List of BPs and the lanes that cover their path

Basic planner	*Lane its path overlap*
FL_BP	{Lane being followed}
CL_BP	{Present lane, New lane}
TL_BP	{Present lane, Traffic lane}
UT_BP	{Present lane, New lane after U-turn}
IT_BP	{Lane entering the intersection}

Supervisor module set the speed limits of BP's planned path using the *set_path_speed* () function provided by the BP's common interface class.

It is undesirable for an AGV to drive and stop in the middle of an intersection region in order to maintain a minimum safety distance d_{safety} from an obstacle. To prevent such unsafe situation, after walking back by safety distance (d_{safety}) along the path, the Supervisor module checks with the BP on whose path *stop_point* falls, to see if it is safe for the AGV can stop in between its path. This is performed using the common interface function *can_stop_along_the_path* (). If the BP does not want the AGV to stop in the middle of its path, the Supervisor module inform the BP to set the beginning of its path (P_{BP}) to zero speed. This for IT_BP causes the AGV to stop before entering the intersection region.

7.5 Check for Dynamic Obstacles

Traffic vehicles that are moving or that might move in the near future, such as a car at a stop sign, are classified as dynamic obstacles. The dynamic obstacles that fall within a zone are handled by zone-related path planners. The dynamic obstacles that fall within a segment or intersection are handled here. The dynamic obstacles that do not fall within any lane or zone are ignored. The system can be updated to handle the dynamic obstacles which are not presently within the lane or zone, but are projected to enter one. This possible extension is discussed in the section 7.5.3.1.

Dynamic obstacles are handled by modifying the speed limits of the already planned path to avoid interaction with obstacles. Because these obstacles are not permanent on the path, it is assumed that there is no need to modify the Basic Planner (BP) sequence or modify the goal of any of the BPs in the sequence.. This assumption may not always be true and a possible update to handle such situations is explained in the section 7.5.3.2. By modifying the speed limits along the planned path, the system can exhibit behaviors such as convoying and stopping to avoid a collision.

Handling dynamic obstacles require first detecting the dynamic obstacles that might pose a danger and then avoiding them. The rest of this section and Section 7.5.1 will discuss how to detect the dynamic obstacles of interest, followed by Section 0 which describes how to handle them.

Detection of all the dynamic obstacles of concern is done in two steps. a) Detect dynamic obstacles that are already on the path. b) Detect dynamic obstacle that would enter the path in future.

The former step is performed similar to checking for static obstacles on the path, except for calling the *on_path_obstacles* (σ, DYNAMIC) function to check for dynamic obstacles. Upon detecting a dynamic obstacle already on the path, the speed limits on the path are updated to maintain a speed equal to the speed of the dynamic obstacle at a *safety_distance* behind the obstacle. The following section explains the second step in detail.

7.5.1 Detect Dynamic Obstacles Entering the Path

Dynamic obstacles can always change their speeds or direction of travel over time. This makes it impossible to accurately project their path and detect obstacles of concern. In order to simplify the detection process, following assumptions about the dynamic obstacles are made:

a) The obstacles will follow traffic rules. For example, if a dynamic obstacle is observed on a lane with a stop sign, it is assumed that it will come and stop at the stop sign and follow intersection precedence.

b) The obstacles will continue to travel at their present speed, unless they are expected to stop as in the above mentioned scenario of approaching a stop sign.

From the first assumption, the traffic can enter onto the present continuous path only through a finite set of points along the path, called the *Traffic Entry Point*s (TE_PT). These traffic entry points depend on where the path lies on the RNDF, which again depends on the Basic Planner (BP) type. Hence, Supervisor module iterates through each BP in the BP sequence, queries for the traffic entry points (TE_PTs) on their paths and checks if there are any dynamic obstacles expected to cross through any of these TP_PTs over the time window of interest. The BPs provide this list of TE_PTs through the common interface function *get_TE_PT* () provided by the BP interface class.

The rest of the section introduces these TE_PTs for each of the five on-segment and transition BPs followed by introducing how these TE_PTs are used to detect dynamic obstacles of concern.

a) **Determine Traffic Entry Points (TE_PTs):** TE_PTs of the three BPs: CL_BP, TL_BP, and UT_BP are described in Figure 43. TE_PTs for these BPs do not change from one instance of the BP to another unlike TE_PTs for IT_BP's path varies from one instance of IT_BP to another.

    Figure 42 provides the algorithm to determine this list of TE_PTs for an IT_BP. Here, PL_{it} is the list of exit waypoints on the lanes passing (no stop sign) through the intersection on which IT_BP is planning a path. ex_wp_{BP} -> en_wp_{BP} represents the exit-entry pair that the IT_BP is planning a path for. PL_{it} does not include ex_wp_{BP} of IT_BP. For each *ex_wp* from PL_{it}, if a path to any of its *en_wp* pairs crosses the ex_wp_{BP}-> en_wp_{BP} path of the IT_BP, then the *ex_wp* is considered as a traffic entry point to look for and is added to TE_PT.

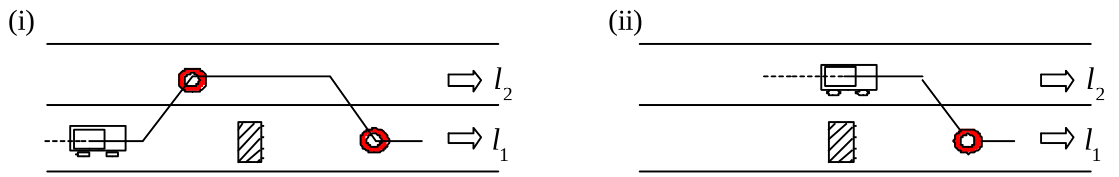
Figure 44 shows the TE_PTs for an example IT_BP. Here it is assumed that none of the exit waypoints have a stop sign and hence $PL_{\mathrm{it}} = \{ex\_wp_1, ex\_wp_2\}$. For the exit waypoint ex_wp_1, the path to its entry pair en_wp_{BP} crosses the path of ex_wp_{BP} ->en_wp_{BP}, as shown in Figure 44 (b), hence ex_wp_1 is added to TE_PT. On the other hand, for the exit waypoint ex_wp_2, paths to none of its entry pairs shown in Figure 44 (c) crosses the path of ex_wp_{BP}->en_wp_{BP}, hence is not added to TE_PT.

FL_BPs do not check for dynamic obstacles driving on to their path. This is

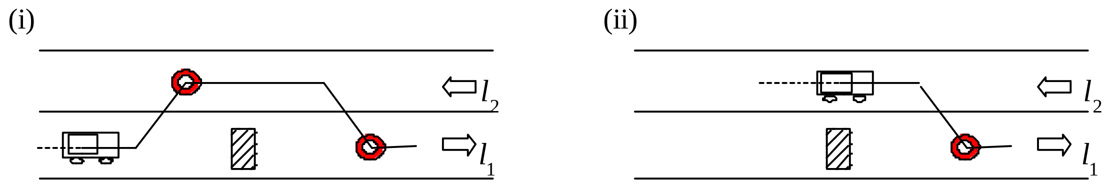
```
Variables:
//Input: $PL_{it}$ - List of exit waypoints of the lanes passing (no stop sign) through the
intersection, excluding the IT_BP's exit waypoint.
//Output: TE_PT, list of traffic entry points
//Here,  $ex\_wp_{BP}$->$en\_bp_{BP}$ - Exit-entry waypoint pair for the intersection Basic
Planner (IT_BP)
for each ex_wp in $PL_{it}$
begin
    for each en_wp pair of ex_wp
    begin
        if (en_wp ->ex_wp and $ex\_wp_{BP}$->$en\_wp_{BP}$ cross)
         then
            add ex_wp to TE_PT (traffic entry point) list
            break
         endif
    end
end
end
```

Figure 42: Pseudo code for extracting TE_PTs on an IT_BP's path

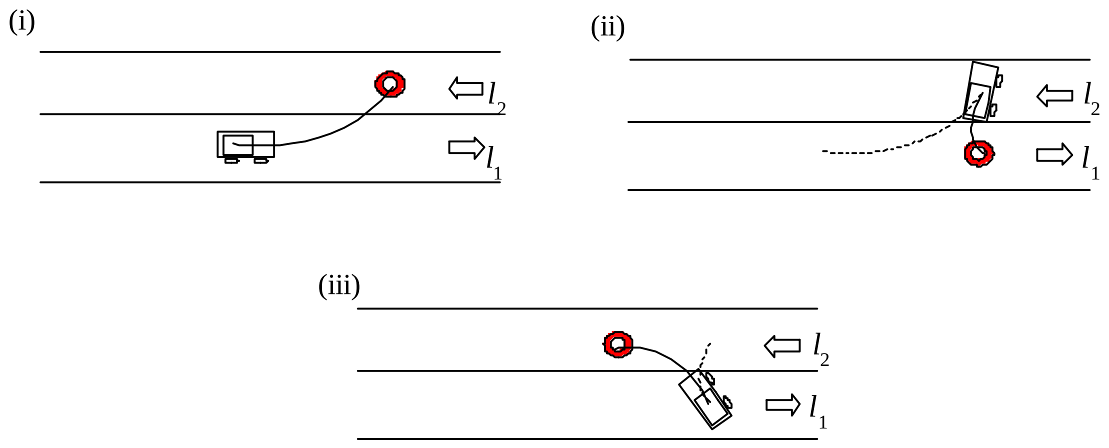
because a FL_BP's path covers a single lane with no traffic lane crossing and having right of way. The only TE_PT for such a path would be at the beginning of the path on the lane. In the BP sequence, if the FL_BP is the first TP, then the AGV is already on the lane and does not need to try to avoid dynamic obstacles approaching from behind. And if the FL_BP is not the first TP, it is always preceded by one of the above four TPs, namely CL_BP, TL_BP, UT_BP, or IT_BP, which already looks for the traffic approaching the lane.



(a) TE_PTs for CL_BP at different phases: (i) before entering neighboring lane, (ii) when reached on the neighboring lane.



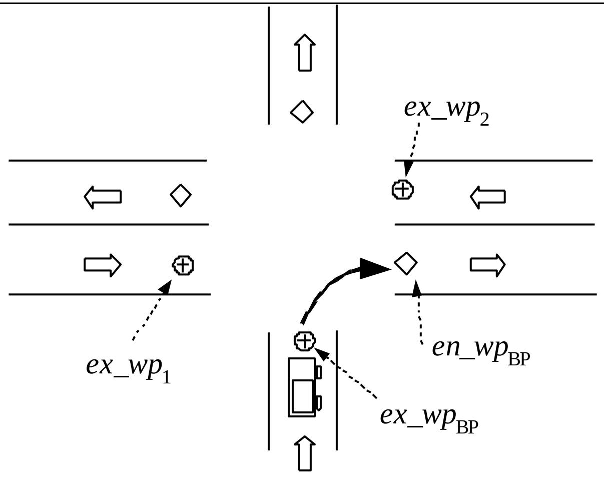
(b) TE_PTs for TL_BP at different phases: (i) before entering traffic lane, (ii) in traffic lane.



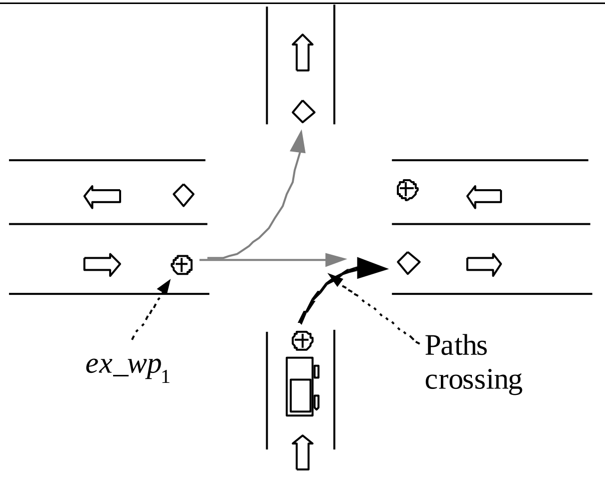
(c) TE_PTs for UT_BP at three phases of U-turn.

Notation: ● TE_PT

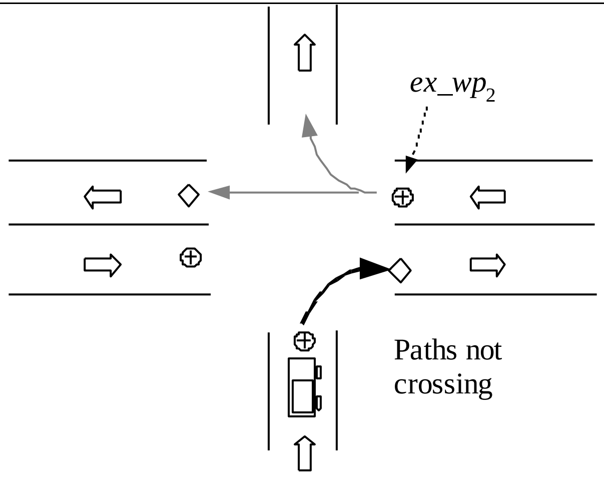
Figure 43: Listing the TE_PTs for CL_BP, TL_BP and UT_BP at different phases of their execution



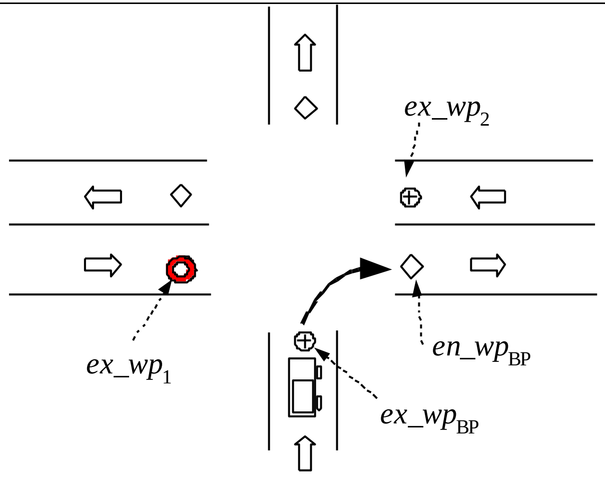
(a) IT_BP scenario ex_wp_{BP} -> en_wp_{TP}, with $PL_{it} = \{ex\_wp_1, ex\_wp_2\}$, two other entry points to the intersection.



(b) Possible intersection paths from ex_wp_1. One of the intersection paths intersects with the *it_bp*'s path.



(c) Possible intersection paths from ex_wp_2. None of its intersection paths intersect *it_bp*'s path.



(d) TE_PT for the Basic Planner includes $\{ex\_wp_2\}$, shown with different wp markings.

Waypoint point notation:

⊕	ex_wp
◇	en_wp
●	ex_wp classified as TE_PT

Figure 44: Listing the TE_PTs for an example *it_bp*

```
//Function Name: handle_dynamic_obstacles ()
//Input: $\sigma$, sequence of BPs

// Handling dynamic obstacles already on the path
for (i=1 to | $\sigma$ |)
begin
    bp = $\sigma[i]$
    // Check for safety against dynamic obstacles already on the path
    (bp, $i_{path}$, obstacle_speed) = on_path_obstacles ($\sigma$, DYNAMIC)
    if (bp != NULL)                    // If found bp with an obstacle on its path at
    then                               //      $i_{path}$ index of its path.
       set_safety_speed_limits ($\sigma$, bp, $i_{path}$, obstacle_speed)
    endif

// Handling dynamic obstacles that might enter path in future
$t_{start}$ = get_time()
for (i=1 to | $\sigma$ |)
begin
    bp = $\sigma[i]$
    TE_PT = bp.get_TE_PT ();
    for (k = 1 to | TE_PT|)
    begin
        $t_k$ = estimate_AGV_time_at_TE_PT ($t_{start}$, $P_{bp}$, TE_PT[k])
        obstacle = world_state.get_earliest_obstacles ($t_k - \delta t/2$, $t_k + \delta t/2$, TE_PT[k])
        $i_{path}$ = bp.get_TE_PT_path_index (k) // returns the path index which represents
                                                // TE_PT [k]
        set_safety_speed_limit ($\sigma$, bp, $i_{path}$, 0)
    end
end
```

Figure 45: Pseudo code for handling dynamic obstacles on BP's path: *handle_dynamic_obstacles* ()

b) **Traffic through TE_PT:** The World_State module maintains a list of dynamic obstacles detected on each lane and can predict their positions at any given time (t) in the future. It extends this feature to provide the list of dynamic obstacles that are expected to pass through any given points within a time window.

The Supervisor module uses the knowledge of AGV's present position and the speed limits along the path to estimate the position (p_k) of the AGV at regular time intervals $[t_k = t_{present} + k * \Delta t]$ in the future. It uses this estimated position (p_k) of AGV at regular time intervals (t_k) and the time window of $\{t_k - \delta t/2, t_k + \delta t/2\}$ to get the list of dynamic obstacles that are of concern at time t_k. Here Δt is the time interval at which checking for dynamic obstacles is performed and δt is the suggested safety time window.

7.5.2 Handling the Dynamic Obstacles on the Path

Dynamic obstacles that are already on the path are handled differently from the dynamic obstacles which are expected to pose danger by entering through one of the TE_PT. The dynamic obstacles already on the path are handled similar to handling static obstacles on the path. At a safety distance behind the obstacle along the path, the speed limit is set to be minimum of already existing speed limit of the path and the speed of the dynamic obstacle being handled. This would limit the AGV from approaching the dynamic obstacle any closer than $d_{dynamic\_safety}$ and at speeds less than that of the front obstacle.

The first part of the pseudo code from Figure 46 performs the above handling. Here *on_path_obstacles* (σ, DYNAMIC) returns the BP on which a closest dynamic obstacle is lying, the index to BP's path, i_{path}, where the obstacle intersects and the speed of the obstacle. This information is used to set safety speed limit from the dynamic obstacle using the set_safety_speed_limits (σ, *bp*, $i_{path,}$ obstacle_speed) function.

Dynamic obstacles that are expected to pose danger by entering through one of the TE_PT is handled by setting speed limit of the path at certain distance before the closest TE_PT to zero speed. This will have the AGV to stop before the TE_PT point.

```
//Function Name: handle_dynamic_obstacles ()
//Input: $\sigma$, sequence of BPs

// Handling dynamic obstacles already on the path
for (i=1 to | $\sigma$ |)
begin
    bp = $\sigma[i]$
    // Check for safety against dynamic obstacles already on the path
    (bp, $i_{path,}$ obstacle_speed) = on_path_obstacles ($\sigma$, DYNAMIC)
    if (bp != NULL)
    then
        set_safety_speed_limits ($\sigma$, bp, $i_{path}$, obstacle_speed)
    endif

// Handling dynamic obstacles that might enter path in future
$t_{start}$ = get_time()
for (i=1 to | $\sigma$ |)
begin
    bp = $\sigma[i]$
    TE_PT = bp.get_TE_PT ();
    for (k = 1 to | TE_PT|)
    begin
        $t_k$ = estimate_AGV_time_at_TE_PT ($t_{start}$, $P_{bp}$, TE_PT[k])
        obstacle = world_state.get_earliest_obstacles ($t_k - \delta t/2$, $t_k + \delta t/2$, TE_PT[k])
        $i_{path}$ = bp.get_TE_PT_path_index (k) // returns the path index which represents
                                                 // TE_PT [k]
        set_safety_speed_limit ($\sigma$, bp, $i_{path}$, 0)
    end
end
```

Figure 46: Pseudo code for handling dynamic obstacles on BP's path: *handle_dynamic_obstacles* ()

7.5.3 Possible Updates

The present system is built to address the situations expected during the Urban Challenge, and the system can be updated to handle more situations. The updates related to dynamic obstacles are listed below.

7.5.3.1 Preparing for safety from off-segment dynamic obstacles

The present system does not handle dynamic obstacles, which are not on the RNDF, but may be approaching the path. In the controlled environment of the Urban Challenge, such a scenario may not happen, but it may be required to handle such situations in the real world driving. This could be due to the bad drivers or due to the RNDF not covering all the lanes from the real-world.

For each dynamic obstacle not on the RNDF, its trajectory path can be computed, which is the path it is expected to follow. In the simplest method, this can be computed as the straight line joining the present position of the obstacle and its expected position at time (t_k) in the future, assuming it continues to drive at the present known velocity vector. This path trajectory can be stored in a database and indexed by the IDs of the lanes it intersects. Note that a single trajectory path can cross multiple lanes at multiple locations. Each of these lanes is considered by having multiple entries of trajectory path in all such lanes.

The World_State can be updated to maintain this database of projected dynamic obstacles on each lane, along with the existing stationary and dynamic obstacles. With this update to the World_State, the present Supervisor should be able handle dynamic obstacle that are off-segment.

7.5.3.2 Handle dynamic obstacles by changing plan

As mentioned earlier, the present system will not change the Basic Planner (BP) sequence or goal of any BP for a dynamic obstacle because the environment is changing. However, there could be some situations where it is better to change the plan for the dynamic obstacles that are detected. For example, if convoying a very slow moving vehicle for long time, it is preferred to change lanes and pass the leading vehicle instead of traveling at low speeds.

The present system's architecture can be updated to exhibit such a behavior. This is achieved by modifying the Supervisor module's main loop to check for dynamic obstacles along with static obstacles and to inform the Re-Planner module of any detected concern. The Re-Planner module needs to be updated with new Behavioral States to handle the concerns from dynamic obstacles.

7.6 Collect Path

In this step, the Supervisor module iterates through the Basic Planner (BP) sequence to collect their planned paths (P_{BP}). This section explains the *collect_path* () function used in Figure 35. These path segments are appended to form one single steering path. This process of iterating through the BP sequence can be terminated for the same four reasons as applied while iterating to trigger the BPs to plan their paths as described in Section 7.3. Generating the final path involves taking care of the proper stitching of the path segments (P_{BP}), and setting consistent speed limits for the final path. The following sub-sections explain each of these two tasks.

7.6.1 Stitching the Path Segments

For proper stitching of steering path segments from consecutive Basic Planner (BP) in the sequence it is required to have an overlapping common point between their path segments. Figure 47 shows an example scenario of path stitching. Here, the points {{st_p$_1^2$}, {st_p$_2^1$}} are overlapping and so are the points {{st_p$_2^2$}, st_p$_3^1$}}. The consecutive BPs have overlapping points so that, depending on the situation, either or both of the BPs may want to control the speed limit at this common point.

The former of the two BPs may have to stop the AGV at the end of its path segment (P_{BP}), at which case it wants to set a zero speed limit for the last point on P_{BP}. Similarly, the latter of the two BPs may need to make the AGV stop before entering its path segment (P_{BP}). For example, an IT_BP may want an AGV to stop before entering the intersection region.

Stitching of the path segments is performed by merging the overlapping points. As both points share the same GPS position, merging of the points involves creating a new path with GPS position set to the common GPS position and setting its speed limit to be a minimum of two points. If $SPS_1$and SPS_2 represent two consecutive steering path segments, and SPS_{final} represents the final merged steering path, then the speed limit of the common point is set as:

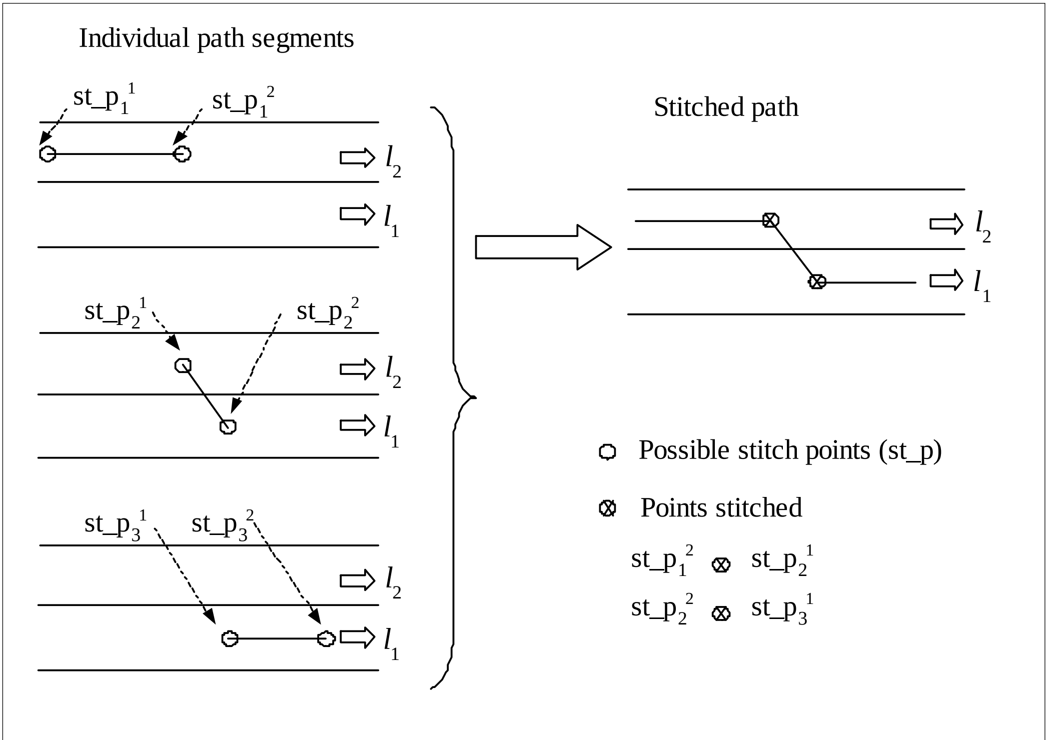


Figure 47: Illustration of path segments stitching

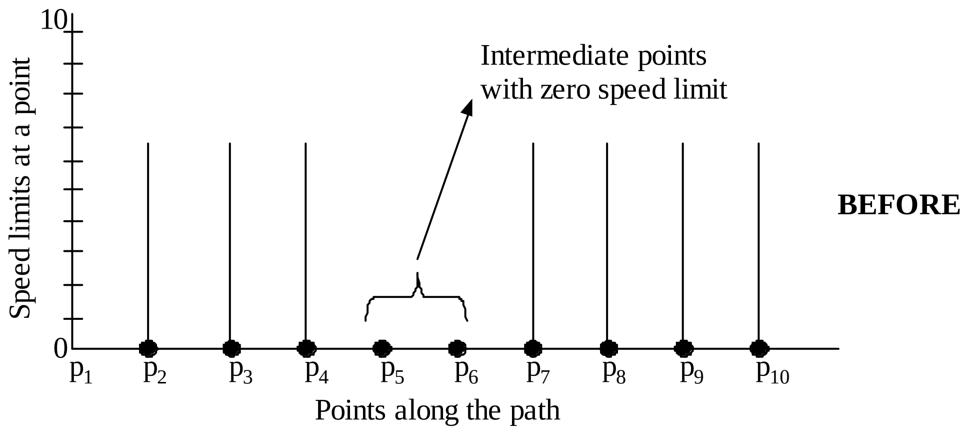
$$SPS_{final}\,[i].\max\_speed = \ min\ (SPS_1[last\_point].\max\_speed, SPS_2[0].\max\_speed)$$

7.6.2 Setting Consistent Speed Limits

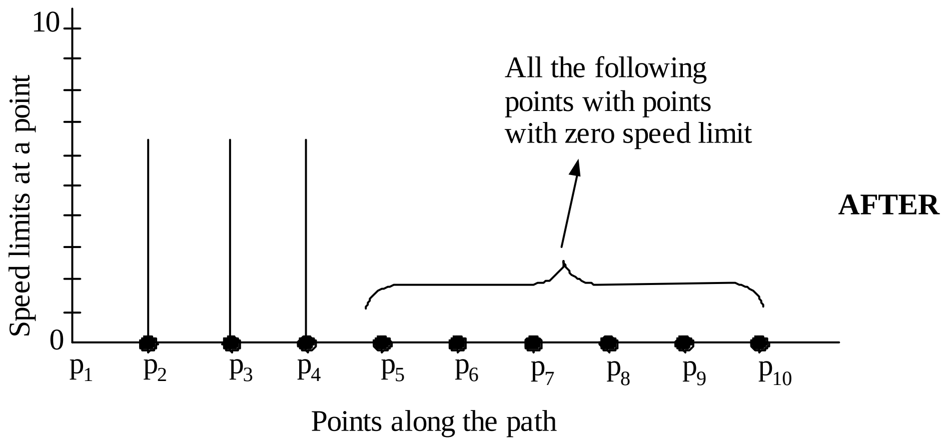
After appending the path segments, the Supervisor walks through the final steering path to make sure that the speed limits are consistent along the path. Consistent speed limits imply that if any point along the path has zero speed limit, then the remaining portion of the path should also have a zero speed limit. Figure 48 illustrates the setting of consistent speed limits for a sample path. It might be expected that the Steering Controller module will stop the AGV before or at the first point with zero speed, so the speed limits of the points that follow should not matter.

The following reasons listed explain how it is safe to rely on this assumption.

a) Upon reaching the point with zero speed limits, the Steering Controller module may look forward at the next point along the path and if this point has a non-zero speed limit, the Steering Controller may accelerate the AGV to drive to this next point.
b) Even if two consecutive points in the path are set to zero speed limits while solving the above situation, the Steering Controller may not be accurate in stopping the vehicle within this short distance.
c) Even if everything works out as expected and the AGV stops at the point with zero speed limit, upon stopping for a small amount of time, there could be slow GPS drift (sensor limitation). The GPS points may shift and the system can think it has crossed the point with zero speed limits, at which point the Steering Controller may start following the rest of the points along the path with non-zero speed limits.



(a) Example path with intermediate zero speed limit points.



(b) Same path after passing through consistent speed limit pass.

Figure 48: Illustration of setting consistent speed limits for a path

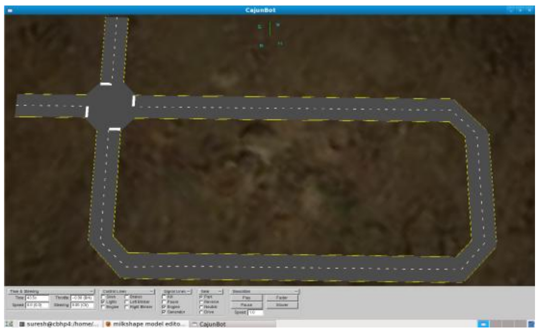
8 Evaluation

This section evaluates the present CB_PP system by providing some notes on the testing the system experienced, both in the CajunBot's simulator world and on the CajunBot-II platform. The section also describes the performance of the system at the DARPA Urban Challenge. The section then gives a comparison of CB_PP's approach to that of the winner of UC, Boss from Carnegie Mellon University, second prize winner, Junior from Stanford University and Skynet from Cornell University.

8.1 Testing and UC Performance

Utilizing the ability to easily add new capabilities, the CB_PP was developed in a spiral development model (Boehm 1988), wherein new capabilities were added over time. With the addition of each new capability, the CB_PP was rigorously tested both in the simulated world and on the CajunBot-II. The tests emphasized both the new capability added and the overall system performance. The testing showed that none of the new capabilities added affected the existing capabilities, proving the CB_PP's claim of easy addition of new capabilities without affecting the existing capabilities. The system covered thousands of miles in the simulator world and hundreds of miles in controlled testing areas in the real world.

The CB_PP was tested at UC's site visit to interact with traffic at intersections by following stop-sign precedence, merging into moving traffic, and handling lane blockages. Site visit was one of the qualification rounds for the UC conducted on 27th June 2007 for Team CajunBot. Here DARPA's representatives visited team and analyzed the CajunBot's capabilities on a rectangular shaped track as shown in Figure 49 (a). On this track, human driven traffic vehicles are used to simulate traffic to simulate above mentioned urban driving


(a) Site visit RNDF track

(b) CajunBot with traffic vehicles.

Figure 49: Site visit

scenarios, as shown in Figure 49 (b). The CBP_PP system completed all the test scenarios without a glitch in less than half the allocated time, the only team out of 53 participants who can claim such a suave test runs.

The CB_PP was tested at UC's National Qualification Event (NQE), where the test scenarios were divided into three areas labeled as A, B, and C. Figure 50 shows the areal image of these test areas along the test tracks. These tests were conducted at George AFB in Victorville, California. Area A tested the capability to merge and cross heavy traffic. Here, the traffic vehicles drove on the outer circle of a figure-eight track and the AGVs were supposed to cross and merge into traffic by driving on one of the smaller circles of this track. CB_PP successfully navigated CajunBot-II through thirteen laps before stopping due to a sensor failure. Area B tested path searching, intersection precedence, road blockages, and re-planning capabilities and CB_PP performed all of these capabilities within the allocated time. Finally, Area C tested mission planning on a larger scale via a zone driving by avoiding obstacles, parking at a designated parking-spot, and navigating through partially-blocked lanes. The CB_PP successfully demonstrated all of these capabilities, except handling partial-lane blockages.

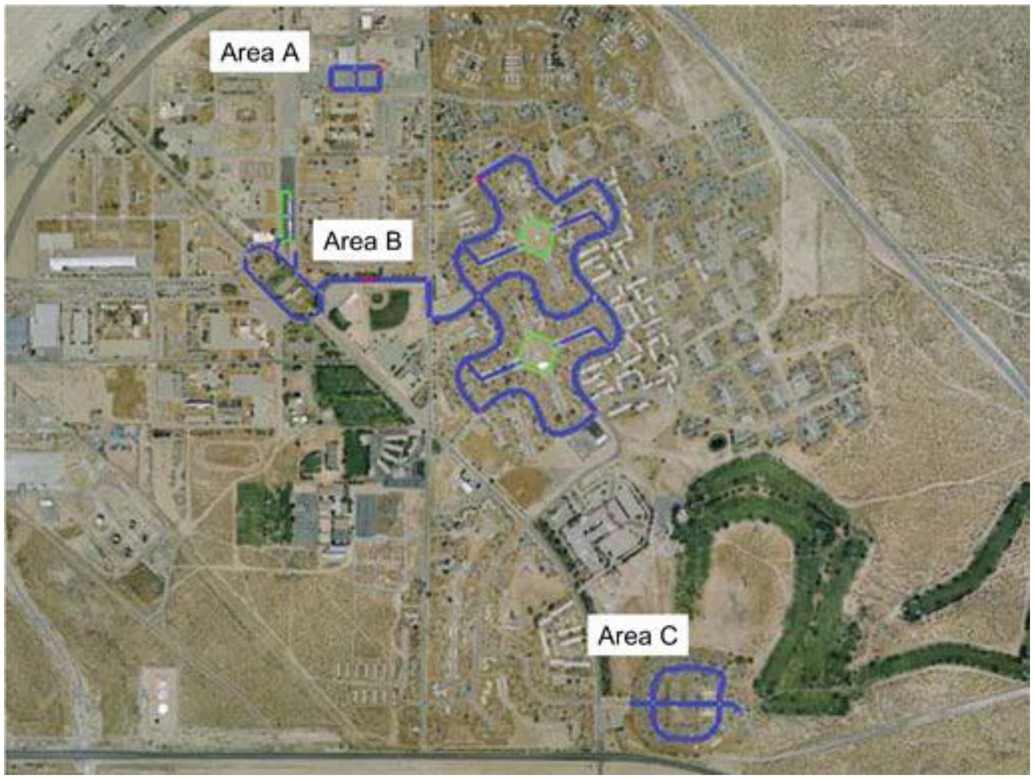


Figure 50: Urban Challenge tracks [Source: (Urmson, et al. 2008)]

In the partial-lane blockage situation, a traffic vehicle is parked on the side of a lane, blocking part of the lane. The AGVs are expected to swerve around the parked vehicle while staying within the lane. This feature was added to the CB_PP just a couple of days before the NQE, and was not tuned and tested for all speeds. This feature, while tuned for only low-speed navigation of 5 mph was run at speeds close to 20 mph in the NQE. For such high-speeds, the CB_PP was not preemptive enough to initiate early swerving and CajunBot-II ultimately touched the edge of a partially-blocking test vehicle. This marked the end of challenge for the team. Though it was disappointing to end the journey, the successful performances at all the other NQE rounds proved that it is not the limitation of the architecture that disqualified the CajunBot-II.

8.2 Comparison with Other Teams

In this section, I compare the present CB_PP's approach with that of the winner of the UC, Boss from Carnegie Mellon University (Ferguson, Howard and Likhachev, Motion Planning in Urban Environments: Part II 2008)(Urmson, et al. 2008); the second prize winner, Junior from Stanford University (Montemerlo, et al. 2008); and Skynet from Cornell University (Miller, et al. 2008). I will first give a brief overview of these three system's approach, followed by a comparison with CB_PP in Section 8.2.4.

To avoid confusion between the different planners, I will refer to each of the planners by their vehicle's name.

8.2.1 Boss (CMU)

Boss performs its planning in three layers, namely, Mission Planning, Behavior Generation and Motion Planning. The Mission Planning represents the input RNDF in a graph representation with directed edges annotated with a cost based on factors such as time of travel, complexity of maneuver, etc. This graph is searched to get minimal cost path to next checkpoint. The graph is updated upon receiving new information about blocked lanes.

The Behavior Generation layer has more focused planning at reduced set of environment constraints. It has three contexts at this layer, namely, Drive Down Road, Handle Intersection, and Achieve Zone Pose. As the name suggests, the first context is used to navigate on a road, second context is used to handle intersection situations such as stop signs and intersection precedence, and the last context is used to perform planning in unstructured environments such as parking-lot, and traffic jam intersection.

Each of these contexts defines rules on how to navigate in the area and how to handle possible situations that may be encountered. The Drive Down Road context explains when to pass vehicles, or when to change lanes and how to handle blocked lanes. The Handle

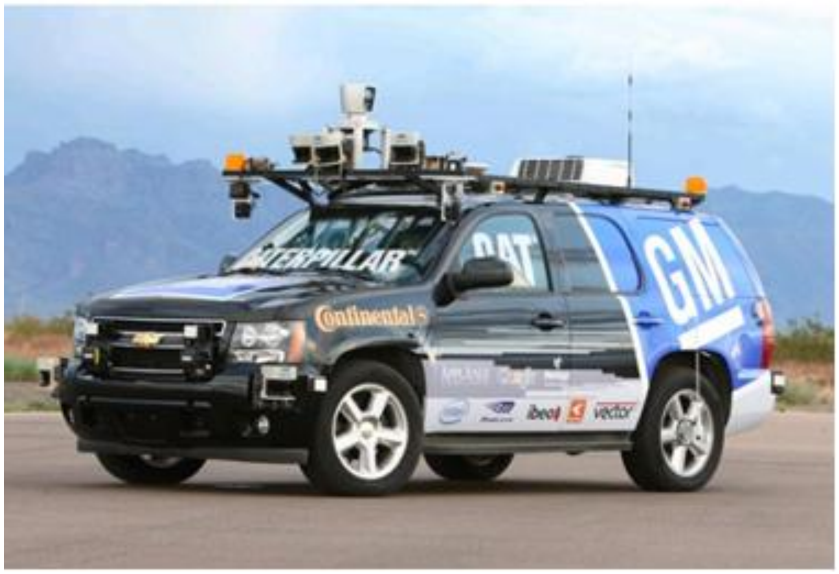


Figure 51: The Boss [Source: www.tartanracing.org]

Intersection context decides when to navigate through an intersection and provides an imaginary lane through the intersection region. The Achieve Zone Pose context defines a goal position and orientation that should be achieved by the vehicle. This context is used to come out of a trouble situation, to a nearest point from which Drive Down Road context can take over.

Motion Planning layer is responsible to drive either on a structured road or in a free-area such as in a zone. While driving on a road, it tries to generate possible vehicle trajectories within the lane and pick the trajectory that drives the vehicle through the center of the lane. It uses trajectory generation algorithm by Howards and Kelly (Howard, Knepper and Kelly 2006) which uses vehicle modeling and forward simulation to predict possible vehicle trajectories.

For free-area driving, as there is no path guidance such as the center of the lane to follow, the free-area uses Anytime D* algorithm (Likhachev, et al. 2005). The Anytime D*

Figure 52: The Junior [Source: cs.stanford.edu/group/roadrunner]

algorithm plans an initial suboptimal plan which is improved overtime upon learning new information about the terrain. This path is used as guidance to explore the possible vehicle trajectories and pick the trajectory closest to this path. This approach is similar to using the center of the lane as guidance when navigating on a road.

8.2.2 Junior (Stanford University)

Junior performs its planning in three layers, namely, Global Path Planning, Behavior Hierarchy and RNDF road and Free-Form Navigation. Global Path Planning layer plans a shortest path from each node in the RNDF to the next checkpoint in the mission, using *Dynamic programming* (Ronald 1960). The cost of each node is computed as minimum of cost to travel to a neighboring node and probability factored cost to reach the goal from neighboring node. Here, the cost to travel to neighboring lane includes penalty for maneuvers that are less preferred, such as a left-turn through traffic.

At the second layer, the Behavior Hierarchy layer represented as a Finite-State Machine (FSM) is used to define the behavior of the system. It uses 13 states (Montemerlo, et al.

2008) representing possible vehicle states such as ‘LOCATE_VEHICLE’, ‘FORWARD_DRIVE’, ‘STOP_SIGN_WAIT’, etc. The FSM defines when to perform an u-turn, or when to cross a yellow-line to avoid a blockage, etc. The decisions made by FSM are implemented by invoking one of the two planners in the last layer.

At the last layer, the planner contains two modules, namely, RNDF Road Navigation, and Free-Form Navigation. As the name suggest, RNDF Road Navigation drives the vehicle on structured areas such as roads. It overlays possible trajectories along the lane including the ones parallel to the center of the lane and selects a trajectory to follow. The trajectories parallel to the center of the lane are considered to handle situations such as a partial-lane blockage.

The Free-Form Navigation drives in areas such as parking lots and traffic jam. This planner uses the goal location and a map to come up with a list of possible trajectories. It uses a hybrid A* approach by representing the vehicle in four-dimension (4-D) discrete grid. The four dimensions include (*x, y, ø, dir*). Where *x, y* represent the position, *ø* represent the heading direction and *dir* represents forward or reverse driving direction. Here, the hybrid A* addresses the discretization issue of the A* approach where the continuous world is divided into discrete cells. Hybrid A* allows cells to store continuous coordinates for the vehicle and thus plan a smooth path.

8.2.3 Skynet (Cornell University)

Skynet performs its planning in three layers, namely, Behavioral Layer, Tactical Layer and Operational Layer. The Behavioral Layer uses the RNDF description and environment information such as lane blockage to plan a path to reach the next checkpoint in the mission.

Similar to CajunBot and Boss's approach, this planner represents the RNDF in a graph representation with initial traversal time cost and dynamic traversal cost. Initial traversal cost is based on elements such as, the length of the trajectory, speed limits, and stop signs. Dynamic traversal cost which decays over time is based on information such as road blockages. At each path planning cycle, Behavioral Layer determines which behavioral state to be used by the tactical layer for planning a path. For Urban Challenge, the system had four states, namely, road, intersection, zone and blockage.

Figure 53: Skynet Vehicle [Source: www.cornellracing.com]

The tactical layer has four components to match the four states of the Behavioral Layer. It switches to corresponding component when the behavioral layer switches between its states. Each component of this tactical layer divides the region around the vehicle into mutually exclusive regions. The first tactical component is the road tactical, which drives the vehicle on a road. It monitors other agents on the road and makes decisions such as, when to perform a passing maneuver. These decisions are made using a decision tree that is built through simulation.

The second tactical component, intersection tactical is responsible for navigating the vehicle through intersection region. It exhibits tasks such as intersection precedence, and intersection queuing.

The third tactical component, zone tactical is responsible for navigating in unstructured regions such as a parking-lot. This component also achieves parking and un-parking maneuvers. This component operates by trying to navigate on human-annotated graphs in the region. These artificial lanes are annotated during preprocessing time. The tactical component uses these artificial lanes to treat zones just like another set of lanes. The lanes to navigate are selected in behavioral layer using an A* algorithm.

The fourth tactical component, blockage tactical is responsible for detecting and navigating the vehicle when forward progress is impossible due to obstacle situation. The operational layer informs the blockage component of possible forward and reverse navigations. This component then confirms if the blockage is not temporary by waiting for multiple planning cycles. If confirmed as a permanent blockage, the component assigns a large time penalty to encourage and search for an alternative route to next checkpoint. This penalty decays over time to allow exploring the route if all other routes are detected to be blocked over the time. If no alternative route is available, the system resets all blockages in order to handle any false blockages detected.

The tactical layer provides their plan as bounding boxes with reference speeds. The Operational Layer navigates the vehicle in this region avoiding obstacles by generating steering, throttle and brake commands. To avoid obstacles, this layer registers the convex hull representing the obstacles in its vehicle-fixed occupancy grid (Ferguson, Stentz and

Table 11: Comparison of Path Planners

Layer	Boss, Junior, and Skynet's Implementation	CajunBot Planner's Equivalent module
Top-Layer	Quickest route for the mission	HLP (High-level planner
Middle-Layer	Plan for immediate goal handle lane-blockage, etc.	Supervisor, Re-planner
Lower-Layer	Generate path to follow	Basic Planners

Thrun, PAO* for Planning with Hidden State 2004). A* search algorithm is used to generate a shortest path around the obstacles. This path is used to guide a non-linear trajectory optimization algorithm. This algorithm tries to smooth the path into a drivable path under the vehicle constraints and obstacles to avoid.

8.2.4 Comparison with CB_PP

All three planners described above used a similar approach of dividing the path planning into three layers. The planners had different naming conventions for these layers, along with minor variations in the division of responsibilities between the layers. To avoid confusion between the naming conventions used by each team, I will refer to these layers as the top, middle, and lower layers of each planner, as shown in Table 11. The top layer of the planner creates the graph representation of the route network and searches for the quickest route, as performed by CB_PP's High-Level Planner module. The middle layer selects the immediate goals to achieve, such as what point along the lane to follow. This layer also makes decisions for situations such as lane blockages. This layer is analogous to CB_PP's Supervisor and Re-Planner modules. Finally, the lower layer of the planner determines the actual path, similar to the Basic Planners (BPs). Unlike CB_PP's concept of pluggable BPs, Boss and Junior use a fixed number (two) of modules at the lower layer to plan paths for all capabilities. One of these modules plans a path for unstructured regions (zones) and the other plans a path for structured regions such as lanes. Both of these modules use trajectory-based planning that explores a list of relevant trajectories and picks the best trajectory as the path to follow.

Having a single module at the lower layer to encompass all capabilities in a broadly divided region provides the planner with the possible advantage of being able to handle situations not explicitly designed for or expected during normal driving. For example, the vehicle could be in a wrong lane and facing off the lane. In such situations, the planner is expected to drive the vehicle to the correct lane and orient itself along the lane, so that the rest of the mission can continue. The CB_PP presently will not be able to handle such unplanned situations, but this is not the limitation of the planner's architecture. CB_PP can be upgraded to handle such situations by having a BP that can plan a path in such situations to safely drive the vehicle from any position to the nearest point on the correct lane, from where the remaining BPs can continue the mission.

On the other hand, having a single module at the lower layer of the planner to provide all the capabilities in a broadly divided region adds more situations that a module must handle, making it difficult to use the individual situation's specifications and have a simpler planning approach. CB_PP makes use of the diversity in the situation's specifications by having a specialized BP for each capability.

Skynet, a finalist of the UC is one of the systems that explicitly had an objective to facilitate expandability of new features, similar to that of CB_PP. Skynet made use of the diversity in the urban driving scenarios by dividing their middle layer into four states: roads, intersections, zones, and blockages.

There are two major differences between this approach and CB_PP's approach to divide planning into multiple BPs. Skynet's planner had division at the middle layer, which is based on the broader definition of the context of driving. On the other hand, CB_PP's division is at the lower layer (BP), based on the capabilities required, in tasks like following a lane and changing lanes. Secondly, Skynet's division at the middle layer allowed the planner to segregate the decision-making rules between contexts, thus simplifying the decision rules. CB_PP has a common set of decision rules that lays above all the BPs in the Re-Planner module.

9 Conclusions and Future Work

With open-ended scenarios that an AGV need to handle for urban driving, the capabilities that the AGV's Path Planner should provide are open-ended. This demanded for an architecture that allows easy plug-ability of new capabilities without affecting the existing capabilities. In the present dissertation, I proposed a Path Planner architecture that allows this easy plug-ability of new capabilities.

In this architecture, each basic capability is provided by an individual Basic Planner (BP), and a sequence of such BPs together achieves the complete mission. A BP interacts with either other BPs or with the three other modules of the Path Planner, namely HLP, Supervisor and Re-Planner. Each of these communications is designed to be transparent of the BP's type. A BP's interaction with other BP is performed using a Base-Path protocol which provides transparency of the BP's type. The HLP's module's only step, namely the BP Extraction that interacts with the BPs, is designed using a logic-based planning or a decision-tree approach, each of which has minimal modification for addition of new BP. For Supervisor module, all its communications with BP is performed via the common interface methods provided by BP_interface class. Finally, the Re-planner module's state-machine representation has each state representing a Behavioral Planner (BhP), allows easy addition of new BhP by adding a new state to the state-machine. A BhP uses a sequence of BPs to exhibit their capability. Adding a new BP to the Path Planner can introduce a set new BhPs that can intern be easily added to the state-machine representation.

A novel approach for detecting dynamic obstacles of concern is introduced. This approach utilizes the urban road network and the assumption that the dynamic obstacles follow traffic rules to simplify the detect ion of dynamic obstacles of concern. The system check for the dynamic obstacles already on the path or are expected to enter the path through a small number of traffic-entry points on the path, within a small time-window.

A novel approach for zone path planning is introduced. This approach merges two traditional approaches, namely the grid-based path planning and Dubin's based path planning. The proposed planner plans a smooth path, as achieved by Dubin's approach at an efficiency that is similar to a grid-based path planning.

I believe the present architecture has lot of scope for improving. Some of which include, being able to handle dynamic obstacles or the situations within a zone, by changing the plan being followed. This work will be continued as future work.

The zone navigation problem is solved in multiple steps, with each step emphasizing on one element of the problem and the results of this step being used as heuristics for the following steps. As the first step, the Zone-Navigator emphasized on obstacles and on efficient path guidance, without considering navigability of the vehicle. This is performed using a grid representation and computing a Gradient-Distance-Field (GDF). This resulting GDF is then used as heuristics for the next step, which emphasizes on the navigability of the path by exploring Dubin's reachability tree.

I believe, this approach can be generalized to solve complex problems by studying all the elements of the problem. The elements of the problem can be segregated and solved in steps, with each step emphasizing on one or more elements. The results of one step can then be used as heuristics for following steps. This approach will need some study on the efficient segregation of the problem elements, and the order in which to solve the problem with the different combination of these segregated elements.

Bibliography

(DARPA), Defense Advanced Research Project Agency. *Urban Challenge.* March 14, 2007. www.darpa.mil/grandchallenge/docs/rndf_mdf_formats_031407.pdf (accessed November 17, 2009).

Agarwal, Pankaj K, Raghavan Prabhakar, and Tamaki Hisao. "Motion Planning for a Steering-Constrained Robot Through Moderate Obstacles." *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (STOC '95).* Las Vegas, USA, 1995. 343--352.

Barraquand, J, B Langlois, and J.-C Latombe. "Numerical Potential Field Techniques for Robot Path Planning." *Proceedings of Fifth International Conference on Advanced Robotics: Robots in Unstructured Environments (ICAR'91).* Pisa, Italy, 1991. 1012-1017.

Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles." *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD'90).* New York, USA, 1990. 322-331.

Bernhard, Seeger, and Hans-Peter Kriegel. "The Buddy Tree: An Efficient and Robust Access Method for Spatial Database." *Proceedings of the Sixteenth International Conference on Very Large Databases.* Brisbane, Australia: Morgan Kaufmann Publishers Inc., 1990. 590-601.

Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *Computer* (ACM SIGSOFT Software Engineering Notes) 11 (August 1988): 61-72.

CajunBot Lab. 2009. www.cajunbot.org (accessed November 17, 2009).

Cline, David, and Parris K. Egbert. "Terrain Decimation Through Quadtree Morphing." *IEEE Transactions on Visualization and Computer Graphics* (IEEE Educational Activities Department) 7 (January-March 2001): 62-69.

Curtis, Andrew Bruce. *Path Planning for Unmanned Air and Ground Vehicles in Urban Environments.* M.S Thesis, Brigham Young University, 2008.

DARPA's Urban Challenge. 2009. www.darpa.mil/grandchallenge (accessed January 2, 2009).

Dubins, L. E. "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents." *American Journal of Mathematics* (The Johns Hopkins University Press) 79 (July 1957): 497-516.

Ferguson, David, Anthony Stentz, and Sebastian Thrun. "PAO* for Planning with Hidden State." *Proceedings of the IEEE 2004 International Conference on Robotics and Automation.* New Orleans, LA, USA, 2004. 2840-2847.

Ferguson, David, Thomas Howard, and Maxim Likhachev. "Motion Planning in Urban Environments: Part II." *Proceedings of the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems.* Nice, France, 2008. 1070-1076.

Finkel, R. A., and J.L. Bentley. "Quad Trees a Data Structure for Retrieval on Composite Keys." *Acta Informatica* (Springer) 4 (March 1974): 1-9.

Flinsenberg, Ingrid Christina Maria. *Route Planning Algorithms.* Netherlands: Technische Universiteit Eindhoven, 2004.

Gutmann, Jens-Steffen, Masaki Fukuchi, and Masahiro Fjita. "Real-Time Path Planning for Humanoid Robot Navigation." *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'05).* Edinburgh, Scotland: Morgan Kaufmann Publishers Inc, 2005. 1232-1238.

Guttman, Antonin. "R-trees: a Dynamic Index Structure for Spatial Searching." *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84).* Boston, Massachusetts: ACM Press, 1984. 47-57.

Herpin, John, Afef Fekih, Suresh Golconda, and Arun Lakhotia. "Steering Control of the Autonomous Vehicle: CajunBot." *AIAA Journal of Aerospace Computing, Information, and Communication (JACIC)* 4 (December 2007): 1134-1142.

Howard, Thomas, Ross Alan Knepper, and Alonzo Kelly. "Constrained Optimization Path Following of Wheeled Robots in Natural Terrain." *Proceedings of the 10th International Symposium on Experimental Robotics (ISER '06).* Rio de Janeiro, Brazil, 2006. 343-352.

La Valle, Steven M. "The Discrete-Time Model." In *Planning Algorithms,* 803-810. Cambridge University Press, 2006.

Lagoudakis, Michail G, and Anthony S Maida. "Neural Maps for Mobile Robot Navigation." *Proceedings of the International Joint Conference on Neural Networks.* Washington D.C, 1999. 2011-2016.

Likhachev, Maxim, David Ferguson, Geoffrey Gordon, Anthony Stentz, and Sebastian Thrun. "Anytime Dynamic A*: An Anytime, Replanning Algorithm." *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).* Monterey, California, 2005. 262--271.

Maida, Anthony S, Suresh Golconda, Pablo Mejia, Arun Lakhotia, and Charles Cavanaugh. "Subgoal-Based Local Navigation and Obstacle Avoidance Using a Grid-Distance Field." *International Journal of Vehicle Autonomous Systems (IJVAS)* (Inder Science) 4 (2006): 122-142.

Miller, Isaac, et al. "Team Cornell's Skynet: Robust Perception and Planning in an Urban Environment." *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part 1* (John Wiley and Sons Ltd.) 25 (August 2008): 493-527.

Montemerlo, Michael, et al. "Junior: The Stanford Entry in the Urban Challenge." *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part II* (John Wiley and Sons Ltd.) 25 (2008): 569-597.

Nilsson, Nils J. "Plan Spaces and Partial-Order Planning." Chap. 22 in *Artificial Intelligence: A New Synthesis*, 385-393. San Francisco: Morgan Kaufmann, 1998.

Patrick, Henry, Winston. *Artificial Intelligence.* Addison Wesley, 1992.

Ronald, Howard A. *Dynamic Programming and Markov Process.* New York: Wiley and Cambridge, MIT Press, 1960.

Russell, Stuart, and Peter Norvig. "Hierarchical Task Network Planning." Chap. 12 in *Artificial Intelligence: A Modern Approach*, 422-430. New Jersey: Prentice Hall, 2003.

Russell, Stuart, and Peter Norvig. "Planning Graph." Chap. 11 in *Artificial Intelligence: A Modern Approach*, 395-402. New Jersey: Prentice Hall, 2003.

Russell, Stuart, and Peter Norvig. "The Planning Problem." Chap. 11 in *Artificial Intelligence: A Modern Approach*, 375-382. New Jersey: Prentice Hall, 2003.

Russelll, Stuart, and Peter Norvig. "Planning with State-Space Search." Chap. 11 in *Artificial Intelligence: A Modern Approach*, 382-386. New Jersey: Prentice Hall, 2003.

Samet, Hanan. "The Quadtree and Related Hierarchical Data Structures." *ACM Computing Surveys (CSUR)* (ACM) 16 (June 1984): 187-260.

Scheuer, A, and Th Fraichard. "Planning continuous-curvature paths for car-like robots." *Proceedings of International Conference on Intelligent Robots and Systems (RSJ'96).* Osaka, Japan, 1996. 1304-1311.

Sellis, Timos, Nick Roussopoulos, and Christos Faloutsos. "The R-tree: A Dynamic Index for Multi-Dimensional Objects." *Proceedings of the 13th VLBD Conference.* Brighton, 1987. 507-518.

Stentz, Anthony. "Optimal and Efficient Path Planning for Unknown and Dynamic Environments." Technical Report, Robotic Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

Stentz, Anthony, and Martial Hebert. "A Complete Navigation System For Goal Acquisition in Unknown Environment." *Preceedings of International Conference on Intelligent Robotic Systems (IROS '95).* Washington. DC, USA, 1995. 425 - 432.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. "A Real-Time Algorithm for Mobile Robot Mapping with Applications to Multi-Robot and 3D Mapping." *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA2000).* San Francisco, California: IEEE, 2000. 321–328.

Treat, J. R., Tumbas, N. S., McDonald, S. T., Shinar, D., Hume, R. D., Mayer, R. E., Stanisfer, R. L. and Castellan, N. J. "Tri-Level Study of the Causes of Traffic Accidents." U.S. Department of Transportation, NHTSA, Washington DC, USA, 1979.

Urban Challenge: Technical specifications. 2009. www.darpa.mil/grandchallenge/docs/Technical_Evaluation_Criteria_031607.pdf (accessed September 30, 2009).

Urmson, Christopher, et al. "Autonomous Driving in Urban Environments: Boss and the Urban Challenge." *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part 1* (John Wiley and Sons Ltd) 25 (June 2008): 425-466.

Venkitakrishnan, Vidhya. "CBWare - Distributed Middleware for Autonomous Ground Vehicles." Master's Thesis, Lafayette, Louisiana, 2006.

Xiaoxi, He, and Chen Leiting. "Path Planning Based on Grid-Potential Fields." *Proceedins of the International Conference on Computer Science and Software Engineering.* Los Alamitos, California: IEEE Computer Society, 2008. 1114-1116.

Golconda, Suresh. Bachelor of Engineering, MJCET, Osmania University, Spring 2002; Master of Science, University of Louisiana at Lafayette, Spring 2005; Doctor of Philosophy, University of Louisiana at Lafayette, Spring 2010
Major: Computer Science
Title of Dissertation: CajunBot Path Planner Architecture for Autonomous Ground Vehicles in an Urban Environment
Dissertation Directors: Dr. Arun Lakhotia, Dr. Anthony Maida
Pages of Dissertation: 145; Words in Abstract: 215

ABSTRACT

This dissertation describes an architecture for a Path Planner system that guides an autonomous ground vehicle through an urban environment while obeying a basic set of traffic rules. As urban driving requires the handling an open-ended set of scenarios, the planner is designed to address the scenarios specified by DARPA’s Urban Challenge (UC); however, the planner permits easy addition of new capabilities to address more scenarios in the future.

Each capability required for urban driving is achieved separately by individual planners called Basic-Planners (BPs). A sequence of these BPs are used by the rest of the Path Planner to achieve the complete mission. The specific type of the BP is kept oblivious from rest of the system to provide easy plug-ability of these BPs. The dissertation introduces the BPs required to match the requirements of UC, and the architecture is designed to provide this plug-ability.

It also introduces a novel approach for path planning in open-areas such as a parking-lot. This planner is created by merging two classical approaches, namely, grid-based planning and Dubin’s based continuous space planning.

The system is then evaluated by specifying its performance at the UC, where system is used to guide CajunBot-II. It then provides a comparison of the present architecture with that of the three of the finalist of UC, namely, Boss, Junior and Skynet.

Biographical Sketch

Suresh Golconda was born on September 4, 1980, in Hyderabad, India. He earned a Bachelor of Engineering in Computer Science from Osmania University (MJCET) in 2002. He then earned a Master of Science in Computer Science from the University of Louisiana at Lafayette in 2005. He will receive a Doctor of Philosophy in Computer Science from the University of Louisiana at Lafayette in 2010.