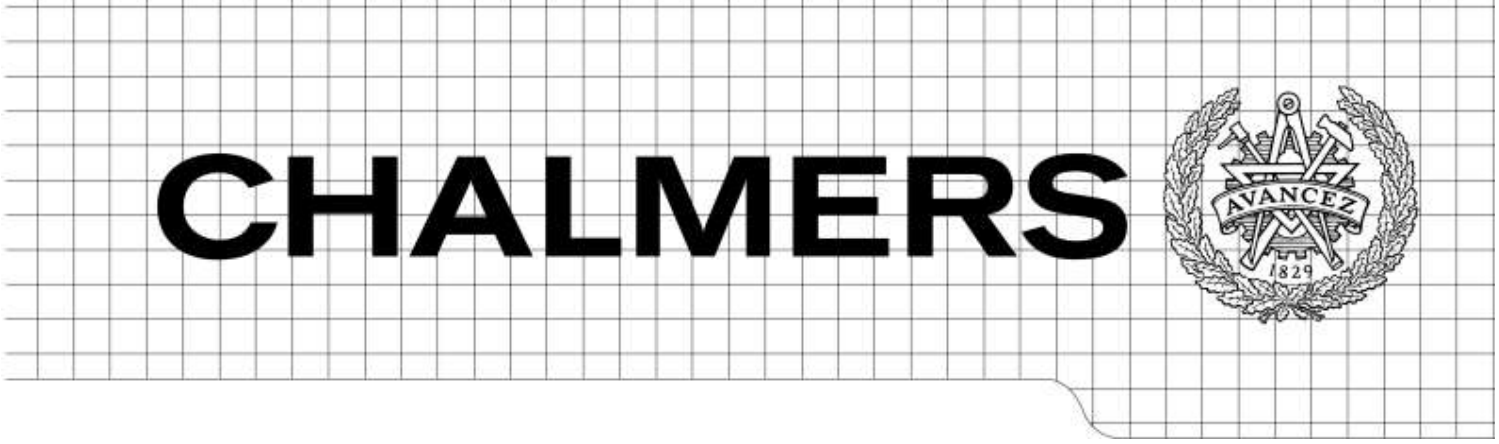


Design and Implementation of Gulliver, a Test-bed for Developing, Demonstrating, and Prototyping Vehicular Systems

Master of Science Thesis in the Programme Networks and Distributed Systems

MITRA PAHLAVAN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2012

Design and Implementation of Gulliver, a Test-bed for Developing, Demonstrating, and Prototyping Vehicular Systems

MITRA PAHLAVAN

Examiner: MARINA PAPATRIANTAFILOU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2012

Table of Contents

List of Figures

Acknowledgment

I am heartily thankful to my supervisor, Elad Michael Schiller, whose encouragement, supervision and support from the preliminary to the concluding level made this thesis possible. I would like to thank Marina Papatriantafilou for the assistance she provided at all levels of the project. Special thanks also to my friend Mohamed Mustafa for sharing the literature, idea and invaluable assistance. Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Preface

This thesis is submitted to fulfill requirements of Master program in Networks and Distributed Systems at Chalmers University, Computer Science and Engineering department. We propose Gulliver as a platform for studying vehicular systems in a large scale open source test-bed of low cost miniature vehicles. This thesis presents the platform, its design and a set of applications that could be demonstrated by Gulliver [1, 2].

Abstract

Vehicular system designers often use simulation tools for proving vehicular systems. The computational complexity of detailed simulations limits the scale of such testings. Therefore, it is often the case that the first full-scale demonstrations of new concepts for vehicular systems are done in proving grounds and testing tracks.

We propose Gulliver as a platform for studying vehicular systems in a large scale open source test-bed of low cost miniature vehicles that use wireless communication and are equipped with onboard sensors. Our approach provides the possibility for a simpler yet detailed investigation of vehicular systems. This thesis presents the platform, its design and a set of applications that could be demonstrated by Gulliver. Gulliver allows the design of vehicular systems to focus on the cyber-physical aspects of the studied problems. Gulliver lies between computer simulation and full-scale vehicle models, and as such, it simplifies and reduces the costs of vehicular system prototyping and development. Note that the cost of each miniature vehicular unit is at least one or two order of magnitude less than a full-scale vehicular prototyping unit.

We expect that Gulliver will allow a wider range of researchers than today to directly contribute to development of future vehicular systems, such as greener transportation initiatives and zero fatality objectives.

After designing the proposed toolkit, the miniature vehicles have been studied in order to shape the design toward the implementing on the test-bed floor. Towards this, we performed extensive experiments with the miniature vehicles. Along with that, we proposed different algorithms for different components of the toolkit in order to use in the simulation and the platform. In addition to the theses proposed algorithms, we designed a *Virtual Traffic Light* algorithm which could be tested by Gulliver as an application of it. Besides, this proposed *Virtual Traffic Light* can be used in Gulliver as a new implementation of the Traffic Light.

1. Introduction

Vehicular systems are expected to ultimately gear vehicles with autopilot capabilities, improve safety, reduce energy consumption, lessen CO_2 omission and simplify the control of traffic congestion. This dramatic change will be the result of advances in driver assistant mechanisms for navigating, congestion control, steering, speed controlling, lane changing, avoiding obstacles to name a few (see Figure 1.1). Moreover, other technologies, such as driverless cars and vehicle platoons, might also appear on the road. Thus, future vehicle systems will be controlled by different types of drivers, i.e., driverless, mechanism assisted drivers and nonassisted ones. We propose to study vehicular systems in a large scale open source test-bed of low cost miniature vehicles that use wireless communication and geared with onboard sensors, such as cameras, laser, radar, speed sensors, etc. Our approach provides a simpler yet detailed investigation of vehicular systems that will be affordable by a wider range of developers than today.

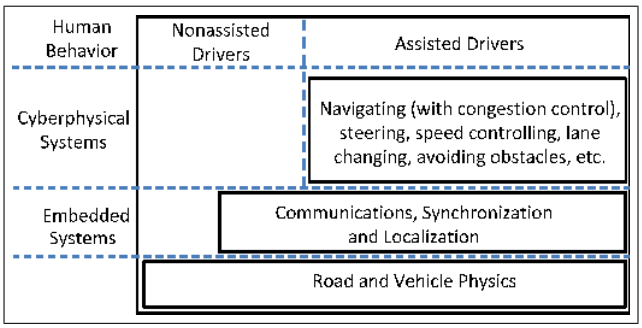


Figure 1.1: Advanced driver assistance mechanisms and their environment

Vehicular system designers often use simulation tools [3–7] for proving new concepts. Simulation tools allow extensive testing of software components, say, by using fault injection methods [8, 9]. Simulators can even deal with complex mathematical modeling of physical objects (e.g., vehicles) and their controlling computer systems (see Figure 1.2). The computational complexity of detailed simulations of future vehicular systems limits the scale of testing and often does not allow extensive system testing for a large number of vehicles. Additional limitations include the absence of human in the loop or the assumption that computer programs can always predict driver reactions.

Due to these limitations, the first demonstrations of new vehicular systems are mostly in proving grounds and testing tracks. DemonstRator [10] is an example of a platform for prototyping

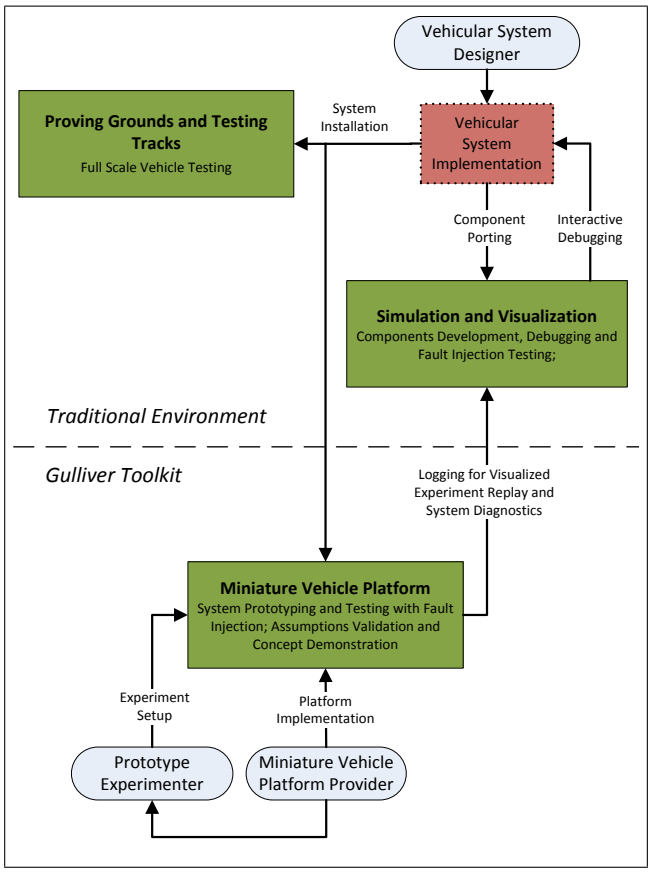


Figure 1.2: Traditional development environment and the Gulliver toolkit are depicted above, and respectively, below the dashed line. The traditional development environment allows the vehicular system designer to simulate and visualize components of the vehicular system before installing it and testing it in proving grounds and testing tracks. The Gulliver toolkit allows the prototype experimenter to use the test-bed for setting up an experiment in which the vehicular system is tested over a miniature vehicle platform. The experiment is logged for later execution visualization by the simulator.

vehicular systems. DemonstRator considers full-scale vehicles, which require isolated testing grounds and considerable protections against vehicle crashing. Proving ground facilities are not affordably accessible to a wide range of universities, public research and engineering institutes. By reducing the demonstration costs, we could allow a greater engineering force to participate in the efforts for greener transportation systems with near zero fatalities.

1.1 Proposed Toolkit

Recent advances in the field of mobile robots allow the ad hoc deployment of a fleet of miniature vehicle that are remotely controlled by human drivers or computer programs. These

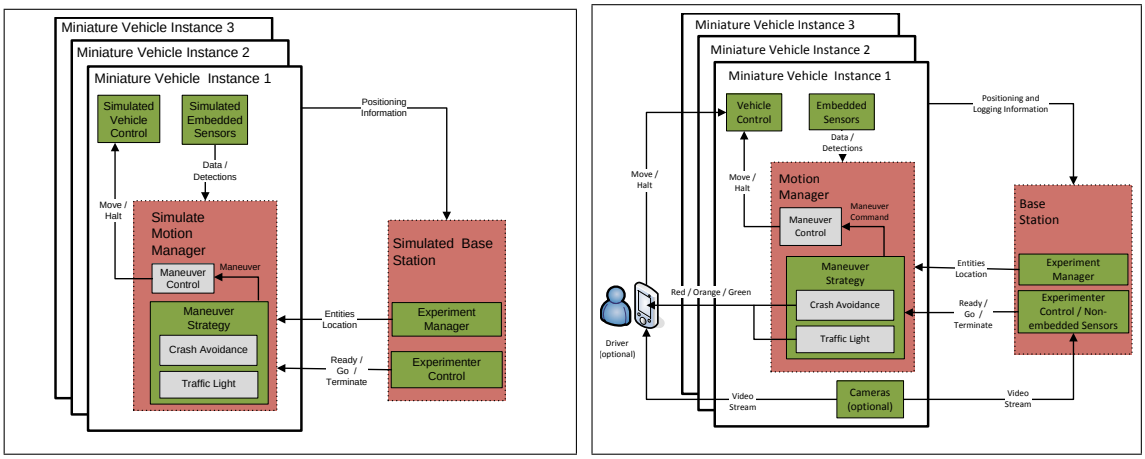
affordable miniature vehicles can greatly simplify the development of the cyber-physical [11] layer of new vehicular systems (see Figure 1.2). Namely, a prototype experimenter can test the vehicle system that is installed on the miniature vehicle platform. These tests can include onboard fault injection. Moreover, the experiment execution can be logged and later replayed and visualized in the simulator.

In order to bring the prototyping of cyber-physical layer into the practical realm, one can take a range of approaches for emulating and substituting relevant system elements. For example, the human driver can be included in the loop of the miniature vehicle control via an onboard or remote computation, hand-held wireless devices, or driver simulation cockpits with multi-angle video streaming in addition to what looks like, sounds like and feels like emulation of the vehicles and their environment.

This thesis focuses on the design of the miniature vehicle platform that we name Gulliver. Gulliver's greatest strength lies in its ability to prototype cyber-physical technologies for vehicular systems. We assume that problems related to the interaction among vehicles to the road can be solved before the prototyping phase (see Figure 1.1). Thus, we can follow the approach in which miniature vehicles can represent full-scale vehicles in the test-bed.

It is up to the prototype experimenter to decide which relevant parts of the embedded system should be included onboard of the vehicle. For example, one of the key difficulties is to understand the impact of vehicle-to-vehicle communication, such as the IEEE `802.11p` standard, which is inherently subject to interferences and disruption. Vehicular systems have safety critical requirements that must mitigate uncertainties, such as the communication related ones. Gulliver can facilitate the study of vehicle-to-vehicle communication at the MAC layer and above, e.g., dynamic bandwidth allocation, pulse synchronization, contention control, packet routing, to name a few. We note the physical layer can also be studied in Gulliver. However, the platform designer should take into account signal shadowing and fading. E.g., the designer can include signal blocking objects in the test-bed area and onboard the miniature vehicle.

Our approach enables the vehicular systems designer to focus on the cyber-physical aspects of problem. Moreover, by including the human driver in the loop, many of the designer assumptions about the cyber-physical system and the human driver can be validated. Thus, the Gulliver test-bed is a multifaceted toolkit for testing, prototyping and demonstrating new vehicular systems. The test-bed feedback capabilities and human interaction units are imperative debugging tools for vehicular system developers.



(a) (b)

Figure 1.2: Component diagrams of the Simulator (a) and the Miniature Vehicle Platform (b)

1.2 Our Contribution

Gulliver presents a multifaceted toolkit for testing vehicular systems in a practical realm. It lies between computer simulation and full-scale vehicle models, and as such, it simplifies and reduces the costs of vehicular system prototyping and development. Gulliver's greatest strength lies in its ability to prototype cyber-physical technologies for vehicular systems. By that, it allows the system designer to focus on cyber-physical aspects of algorithmic problems in vehicular systems and their networks.

In addition to presenting Gulliver as a concept, this thesis outlines the design of its key components. We explain how the miniature vehicles can follow a strategy that allows them to safely traverse the test-bed floor along their Route Plan. The strategy is based on mechanisms for crash avoidance and traffic light signaling (see Section 4).

We report on our implementation efforts. We explain how the prototype experimenter can assure that the miniature vehicles can drive along the lane markings. We also explore additional technological challenges (see Section 5).

Future vehicular systems will enable vehicular interaction, cooperation and will be the first cyber-physical systems to reach the scale of million units. Currently, no safety-critical system comes close to this scale. Gulliver design is the first to facilitate the detailed investigation of the vehicle interaction and emerging patterns among hundreds and even thousands of units of a cyber-physical system. These investigations are imperative for the design and development of advanced driver assistant mechanisms, such as virtual traffic light, vehicle platooning, coordinated contention control, coordinated lane change, to name a few.

2. Related Works

Wireless ad hoc networks are often simulated before their installation, e.g., TOSSIM [12], which is TinyOS mote simulator. Wireless Vehicular Networks (VANETs) are often simulated by systems that have a wireless ad hoc network simulator, and microscopic traffic flow simulator, such as SUMO (Simulation of Urban MObility) [13]. It provides information to the vehicles about how they can traverse along their routes. Similarly, DIVERT (see [14]) is a traffic simulator which models vehicles mobility and their communication. Gulliver extends the use of simulator and for the first time allows testing of new concepts in a test-bed that includes miniature vehicles. The proposed approach allows prototyping of vehicular systems in a more practical realm than computer simulations.

One of the important issues that future vehicular systems will deal with is accident prevention. In [15], the authors describe a self-organizing virtual traffic light (VTL). VTLs allow the ad hoc deployment of traffic lights in every road intersection. The authors of [15] use leader election mechanisms for allowing a single vehicle to serve as the VTL server. It is up to this server to broadcast the VTL's status to arriving vehicles. These messages are then displayed to the drivers. The leader election criterion includes proximity considerations and requires agreement. Their concept is demonstrated via DIVERT [14] with sampled traffic information.

A more robust approach for VTL construction is presented in [16–18] using Virtual Node Layer (VNLayer). VNLayer is a programming abstraction in order to have virtual nodes emulated by physical nodes. They use the VNLayer for emulating the virtual traffic light. The implementation of traffic light in [16, 18] is deployed via a small set of HP iPAQ hand-held computers that are mounted on slow moving robots.

We consider a similar accident prevention service for future vehicular systems, namely a traffic light. However, we demonstrate our concept via extensive testing in a platform that has many miniature vehicles and a logging/replay mechanism, rather by simulation only or a small set of traffic scenarios.

3. Preliminaries

We list the assumptions, definitions and notations that are used in this thesis.

3.1 Road Map, Route Plan and Lane Marking

We consider drivers of miniature vehicles that plan their way using *road maps*, which the prototype experimenter provides. The driver's *route plan* sets the course of travel and assists with the vehicle navigation, e.g., "on the next intersection, turn left!"

The roads include *segments* that have the index set $S = \{ 1, 2, \ldots n\}$. Each segment, $s_i \in \{s_k\}_{k \in S}$, has at least one entrance or exit. We define $in(s_i), out(s_i) \subseteq S$ as the sets that include s_i's entrances, and respectively, exits. We define road intersections as segments that have more than one entrance. A *road map* is a directed graph $G(\mathcal{S}, E)$, in which $\mathcal{S} = \{s_k\}_{k \in S}$ is the set of segments (vertices) and $E = \{\langle s_u, s_w \rangle \in \mathcal{S} \times \mathcal{S} : in(s_w) = u \wedge out(s_u) = w\}$ (edges), where s_u is the segment from which the vehicles can enter segment s_w. We note that the prototype experimenter can show the *route plan* to the driver by presenting a directed path that leads from source to destination on the graph $G(\mathcal{S}, E)$. In our pseudo-code, we use the array $RoutePlan_{v_i}[]$ for listing the segments that vehicle v_i should traverse from source, $RoutePlan_{v_i}[s]$, to destination, $RoutePlan_{v_i}[d]$, where $s = 0$ and $d = sizeof(RoutePlan_{v_i}) - 1$.

Given a road map, $G(\mathcal{S}, E)$, we require from the prototype experimenter to define how each segment, $s_i \in \mathcal{S}$, is situated on the test-bed. Lane markings are used to align vehicles on the road, i.e., the driver should steer the vehicle between two parallel dashed lines that are marked on the test-bed floor. We also consider virtual lane marking. Namely, the miniature vehicle could aim at driving along a line in the Euclidean plane that is not marked on the test-bed floor.

We assume that each segment, s_i, has an index set, $L_{s_i} = \{1, 2, \ldots k\}$ that represents lanes. For each lane, $\ell \in L_{s_i}$, we define $in(\ell)$, $out(\ell) \subseteq \mathcal{S}$ as ℓ's entry, and respectively, exit segments in s_i, i.e., $in(\ell) \in in(s_i)$ and $out(\ell) \in out(s_i)$. For the case of $in(s_i) = \emptyset$ or $out(s_i) = \emptyset$, we define $in(\ell) = s_i$, and respectively, $out(\ell) = s_i$. Let $s_f \in in(s_i)$ and $s_t \in out(s_i)$. We define $FT_{s_i}(s_f, s_t) = \{\ell \in L_{s_i} : in(\ell) = s_f \wedge out(\ell) = s_t\}$ as the index set of all lanes in segment s_i that go from s_f to s_t.

The vehicle may access information that is local to its current segment, *CurrentSegment*$_{v_i}$, and its lanes, $\ell \in L_{s_i}$. Namely, $in(s_i)$, $out(s_i)$, $in(\ell)$ and $out(\ell)$. When the vehicle current lane is *CurrentLane*, it can also query information about other vehicles in its proximity and nearby lanes.

We define the set V as the set of system objects (vehicles, pedestrians, etc) and $V(v_i) \subseteq V$ as the set of object that vehicle $v_i \in V$ queries about. This information includes $v_j \in V(v_i)$ current location, $location_{v_j}$. Moreover, vehicle v_i can know if object (vehicle) $v_j \in V(v_i)$ is on its current trajectory. We require that $V(v_i)$ includes all the objects that are on the line of sight with v_i, i.e., any object, v_j, for which there is a straight line of bounded length to v_i that does not cross any object between them. Namely, we assume the existence of a primitive, $OnTrajectory_{v_i}(v_j)$, that indicates whether object (vehicle) v_j is on v_i's trajectory. We note that the aforementioned information can be retrieved with the aid of maps, positioning systems, (virtual) lane markings and additional technologies, such as image recognition, e.g., [19, 20].

For a vehicle $v_i \in V$, we define $position_{v_i}$ that represents v_i's current location, $location_{v_i}$, and heading (the direction in which the vehicle chassis is directed towards).We consider virtual lane as a sequence of vertices, p (path), that are placed along the lane. Each vertex is associated with a geographical location and orientation, which is the desired location and heading that the vehicle should aim when following p. For calculation of the orientation in vertices, suppose that, all neighbor vertices are connected by vectors and orientation in a vertex is estimated by using the average of directions of its vectors, [21]. In Figure 3.1, Orientation in vertex B is calculated by using average of direction of the $\overrightarrow{AB}$ and $\overrightarrow{BC}$ vectors.

The vertices are defined in such a way that a curve, $c(p)$, could connect all of p's vertices and follow their orientations. When a miniature vehicle, $v_i \in V$, is changing its lane p_1 to lane p_2 or returning to its lane (p_1) from a point (point can be defined like a vertex) out of the lane, we consider $position_{v_i}$ and $LaneSuffix(location_{v_i}, lane)$. $LaneSuffix(location, lane)$ is the $lane$'s suffix that starts at nearest point of the $lane$ on $c(p)$ to $position_{v_i}$. In other words, $LaneSuffix(location, lane)$ is the sequence of points which are positioned ahead of the vehicle based on its path.

In both cases, lane changing and lane correction, the vehicle needs to take a maneuver to the target lane. We call the maneuver, *corrective trajectory*, which provides a trajectory from $position_{v_i}$ to the target lane. The *corrective trajectory* should be smooth which means the curvature of the trajectory should be smooth in order to have a feasible steering. Traversing the succeeding points of the $position_{v_i}$ guarantees that the vehicle won't move to the preceding points of the lane and fulfils the safety requirement to some degree for the vehicle's maneuver. The safety requirement refers to the fact that the vehicle should not move back to the lane because it will cause a car crash with the vehicles on the lane. In *corrective trajectory* the vehicle can have a backward movement although it aims to reach the succeeding point, so for having safety, a *correc-*

tive trajectory which has no backward movement will be chosen. In other words, the difference between vehicle's heading and orientation in the goal point (the *corrective trajectory* positions the vehicle at a point on $c(p)$ which is called the target point) should not be more than 90 degrees otherwise the vehicle will have a backward movement despite the goal point being ahead of the vehicle. In Figure 3.2 (a), *corrective trajectories* $1, ..., 4$ involve backward movement.

In some situations, like when the heading of the vehicle is significantly different from the expected heading, the *corrective trajectory* has a backward movement in order to correct the vehicle's lane, see Figure 3.2 (b). In such scenarios, the *corrective trajectory* which positions the vehicle at the nearest succeeding point, will be chosen.

By traversing a *corrective trajectory*, the vehicle meets the target lane at a point with a specific heading and cross track error. The best point on the target lane should be found in order to move with a safe and smooth maneuver. Finding the best point on the target lane is based on the proposed approach in [21, 22]. The $ProposeManeuver(position_{v_i}, LaneSuffix(location_{v_i}, TargetLane))$ searches over possible points among $LaneSuffix(position_{v_i}, TargetLane)$, and minimizes a utility function which penalizes different constraints like cross track error, curvature, heading error and departure time, and returns the optimal *corrective trajectory* to the lane. Cross track error refers to the distance of vehicle to the path. The (lane) departure time refers to the period during which the miniature vehicle is not following its lane, say, due to unexpected lane departure or intentional lane change. Namely, when the miniature vehicle unexpectedly departures a lane, we consider the time it would take the vehicle to return to its lane from its current location. Moreover, lane change occurs by replacing the vehicle's current lane, with the target lane. Thus, the lane departure time in the latter case considers the total duration it takes to move between lanes. In both cases, we refer to the trajectory that the miniature vehicle follows as a *corrective trajectory*. The departure time for the optimal *corrective trajectory* should not exceed the predefined threshold which specifies the maximum duration for traversing a *corrective trajectory*. If the traversing time for the optimal *corrective trajectory* is more than the threshold, the $ProposeManeuver(position, LaneSuffix(location_{v_i}, TargetLane))$ returns $\perp$.

Primitive $move(maneuver)$ sends the steering parameters to the vehicle in order to have $maneuver$ for traversing the selected *corrective trajectory*.

3.2 Maneuver Control

We assume that the miniature vehicles are able to perform basic maneuvers, such as keeping their lanes ($KeepLane$) and changing their lanes ($ChangeLane$), see [23]. The pseudo-code uses

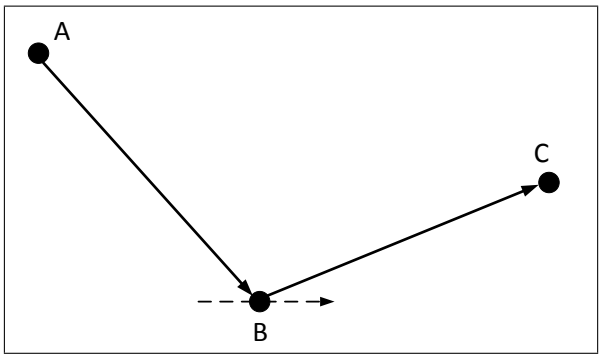
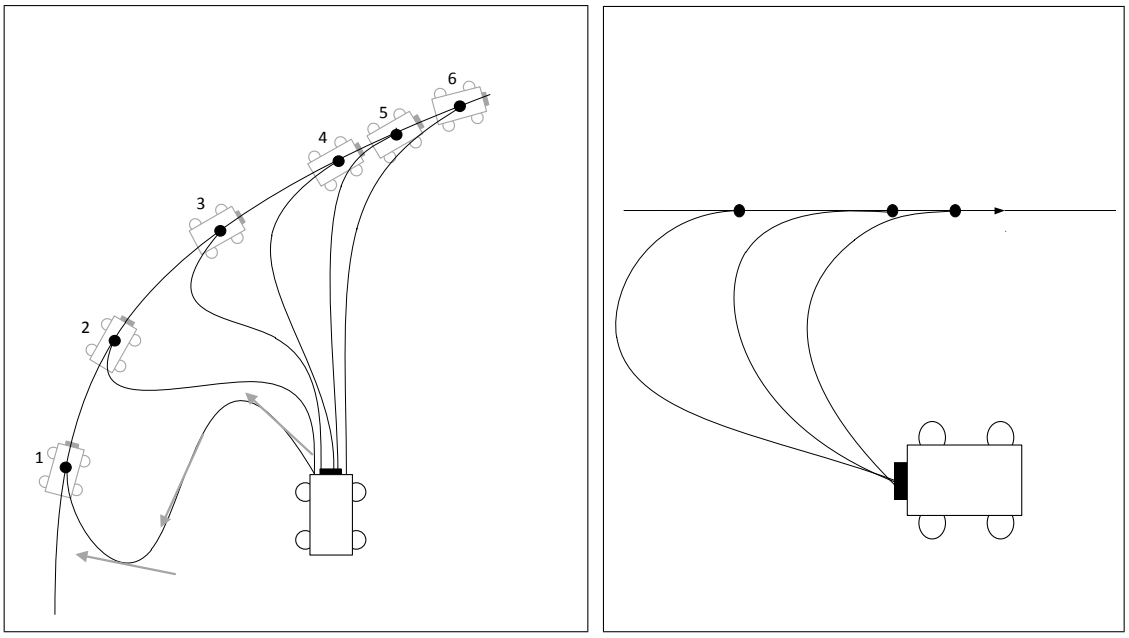


Figure 3.1: Estimation of orientation in vertex B



(a) (b)

Figure 3.2: Different Corrective Trajectories (a) and Corrective Trajectories with backward movement (b)

the primitive $Maneuver(Command)$ for issuing maneuver commands. For example, the vehicle will change its lane when using the command *ChangeLane*, and will instruct the maneuver control to keep the current lane when using the $KeepLane$. The maneuver command $Stop$ halts the vehicle.

Our pseudo-code includes the timer events, $\mathbf{timeout}(Tag)$, where Tag is the timers' name. Timer events are raised by calling the function $SetTimer(Tag, Period)$, where $Period$ specifies the time period after which a $\mathbf{timeout}$ event where Tag is raised.

4. Design Outline

The test-bed includes two subsystems: a Simulator and a Miniature Vehicle Platform (see Figure 1.2). These subsystems both share several components. We first describe the two subsystems before looking into the main component, Motion Manager.

4.1 Simulator

The Simulator subsystem allows the vehicular system designer to develop and test new components (see Figure 1.2 (a)). It has two main components: The Base Station, and the Miniature Vehicle on its many instances.

The Base Station includes the Experimenter Control that can send to the Motion Manager the key platform command, Ready, Go and Terminate, for initializing, starting, and respectively, terminating the experiments. The Base Station also includes the Experiment Manager, which sends to the Motion Manager all the information that is required for moving the miniature vehicles in the platform according to the experiment plan, i.e., Road Map, Route Plan and Virtual Lane Marking. The Base Station monitors and controls the experiment execution by periodically receiving the vehicle's positing information.

The Miniature Vehicle controls the vehicle motion after receiving data from the onboard sensors and commands from the Base Station. The Motion Manager is the unit that controls the vehicle by issuing the commands Move and Halt for steering, and respectively, stopping the vehicle. These two commands are generated by the $Maneuver()$ primitive and allow each vehicle to take a sequence of maneuvers from source to destination along the traveling route.

4.2 Miniature Vehicle Platform

This subsystem allows prototype demonstration and testing of vehicular systems together with its components. During such experiments, the system logs its states for later diagnostics and playback in the simulator (see Figure 1.2). The logging information can be either stored by the vehicle processing units or transmitted on the fly. Future extension of our design can also consider onboard fault injections.

In addition, one can validate the designer assumptions regarding the behavior of the human driver. Our implementation considers hand-held wireless devices (see Figure 1.2 (b) and Section 5). Future extensions can use driver cockpits with multi-angle video streaming in addition to

Algorithm 1: Traffic light mechanism for road intersections

Input $Intersection$: ID of segment in which the traffic light resides;
Local $LightState$: Set of tuples, $\langle Segment, Status \rangle$, where
$Segment \in in(Intersection)$ is an incoming road segment and
$Status \in \{Green, Orange, Red\}$ is a traffic signal for $Segment$;
Local $ActiveIncoming$: Active incoming road segment that is about to receive the GREEN signal, or it just did. We assume that the system initialize to one of the incoming road segments, $in(Intersection)$;
Local $NextIncoming$: Next active incoming road segment;
Constant $GreenPeriod$, $GreenTag$ and $OrangePeriod$, $OrangeTag$: Two pairs of timer periods and tags. These timers are used for assuring that the incoming vehicles have sufficiently long periods for crossing the intersection, and respectively, preparing for the traffic light change. We assume that the system periodically assures that at any time either the timer $GreenTag$ or $OrangeTag$ is set (but never both of them);
Upon timeout$(TimerTag)$
begin
  case $TimerTag = GreenTag$; /* Start the green light period by setting green to the active incoming segment and red to all other segments */
    $LightState \leftarrow \{\langle ActiveIncoming, Green \rangle\} \cup \{\langle Segment, Red \rangle : Segment \in in(Intersection) \setminus \{ActiveIncoming\}\}$;
    $SetTimer(OrangeTag, OrangePeriod)$; /* Set the timer to the next light period, orange */
  case $TimerTag = OrangeTag$; /* Start the orange light period by setting orange to the active and the next incoming segments and red to all other segments */
    $NextIncoming \leftarrow next(in(Intersection), ActiveIncoming)$; /* Find the next incoming segment */
    $LightState \leftarrow \{\langle ActiveIncoming, Orange \rangle, \langle NextIncoming, Orange \rangle\} \cup \{\langle Segment, Red \rangle : Segment \in in(Intersection) \setminus \{ActiveIncoming, NextIncoming\}\}$;
    $ActiveIncoming \leftarrow NextIncoming$; /* Set the active incoming segment */
    $SetTimer(GreenTag, GreenPeriod)$; /* Set the timer to the next light period, green */
Function $TrafficLight(Segment)$
begin
  if $Intersection = Segment \wedge \mid in(Intersection) \mid > 1$ **then**
    return $LightState$; /* When querying $Intersection$'s traffic light, return its state */
  return $\emptyset$; /* Otherwise, indicate that there is no traffic light for this segment */

what looks like, sounds like and feels like emulation of the vehicles and their environment.

4.3 Motion Manager

This key component is mounted on the miniature vehicle and is in charge of receiving sensory information, which includes the vehicle location, and deciding which maneuver the vehicle should take. The maneuvers are controlled by the $Maneuver(Command)$ primitive (see Section 3). It is up to the Maneuver Strategy to decide on which $Command$ each miniature vehicle should take. The Maneuver Strategy must make sure that the miniature vehicle does not crash when traveling to its destination. In order to do that, we use two mechanisms for crash avoidance and traffic light signaling. Next, we present these mechanisms before presenting the Maneuver Strategy itself.

4.3.1 Crash Avoidance Mechanism for Miniature Vehicles

Gulliver considers miniature vehicles that are remotely controlled either by computers or human drivers. Human drivers may ignore the system warning or behave in an unexpected manner. Therefore, there is a need for keeping the equipment safe by using a crash avoidance mechanism.

In current vehicular technology, crash avoidance mechanisms assist actively the driver to avoid accidents by detecting obstacles, pedestrians and other vehicles by using (infrared) vision technology and sensors that are based on laser and radar. Upon detection of the mechanism who may warn the driver, assist the driver in steering and even breaking. In addition, such mechanisms can include adaptive cruise control, lane departure warning systems, to name a few. We do not assume that all of the aforementioned sensory information and mechanisms are accessible onboard the miniature vehicle.

Therefore, we present a (default) crash avoidance mechanism that is based merely on the vehicle location and the *OnTrajectory*$_{v_i}(v_j)$ primitive, see Section 3. We consider the aforementioned sensory information and mechanisms as the basis for possible extensions. The default crash avoidance mechanism for vehicle v_i does not allow the vehicle to get too close to object that resides on its trajectory, see Algorithm 3.

4.3.2 Traffic Light Mechanism for Miniature Vehicles

Traffic lights assure that at any time, no two vehicles from conflicting directions may enter the intersection [see 24]. In the proposed platform, traffic lights are important for assuring that the miniature vehicles do not crash when entering intersections, i.e., a road segment with more than one entry. One can implement the needed traffic light by placing a mote near the intersection

and letting it communicate with arriving vehicles via periodic beacons that encode the traffic light state. This state, of course, needs to be displayed by the human driver units. Another way to go is to let the arriving vehicles communicate among themselves and emulate a *virtual traffic light*, see [15, 16]. We consider both choices of design, but refer to former alternative as the default. We note that the latter design alternative can be implemented by emulating the default traffic light over virtual stationary automaton [17], as in [16].

The proposed base station traffic light is depicted by Algorithm 1. We assume that the base station periodically broadcasts *LightState*, the beacon, to all arriving vehicles. A timer mechanism is used trigger the light changes. The timer considers two periods: green and orange. In the green period the traffic light lets the active incoming segment to receive the green light, while all other road segments get the red signal. The orange period allows the traffic light to safely change the active incoming segment. We iterate over the set of the incoming segments and find the next segment to be active. The traffic light signals orange to both the active segment and the next segment, while all other road segments get the red signal. Both periods end by setting a timer for the next one.

We assume that the vehicles, v_i, cash the beacons received from the base station. Vehicle v_i can query the traffic light state by executing the function $TrafficLight(CurrentSegment_{v_i})$.

4.3.3 Maneuver Strategy

The strategy allows the miniature vehicle to safely traverse the test-bed floor along the Route Plan. We present the maneuver strategy in Algorithm 2. The algorithm consists of a single action, which we assume to be fired periodically.

Before and after taking this update action, the algorithm tests that vehicle v_i satisfies safety conditions, such as crashing avoidance requirements, the traffic light state and the instructions of the prototype experimenter.

The action starts by testing that the vehicle has not reached it destination and that it follows its route plan correctly. Then, the algorithm deterministically selects a lane that leads vehicle v_i to its destination. Different cases of lane selections are considered: (1) No lane in the current segment can lead the vehicle to the next correct segment, (2) The current lane is the correct one, and (3) Vehicle v_i is required to perform a lane change maneuver in order to reach its destination. The latter case requires the algorithm to test safety conditions that concerns the distance between v_i and any other object, v_j, on its current lane or the lane to which it is moving into.

Algorithm 2: Maneuver Strategy for Vehicle v_i

Input $V(v_i)$: Set of vehicles that are on v_i's line of sight;
Input RoutePlan[]: Array of segments that defines v_i's route plan;
Input CurrentSegment: ID of v_i's current segment;
Input CurrentLane: ID of v_i's current lane;
Input $location_{v_i}$: Current geographical location of vehicle v_i;
Side effects $Maneuver(Command)$: Maneuver control primitive. The commands $KeepLane$, $Stop$, and $ChangeLane$ are used for keeping the current lane, stopping, changing to lane;
External TrafficLight(): Red, Orange and Green are the three Traffic Light signals;
External CrashAvoidance(): Red and Green are the two Crash Avoidance signals;
External ExperimenterControl(): Ready, Go and Terminate are the three External Control commands;
Local Cursor: Iterator for v_i's $RoutePlan$. Initialized to the first element, 0;
Local TargetLane: When v_i is changing lane, this is the ID of the lane which v_i will use;
Local distance(locA, locB): Euclidean distance between locations $locA$ and $locB$;
Constant SafeDistance: Minimum required distance between any two vehicles that are moving in different lanes;
Alias PreviousSegment = in(CurrentLane), NextSegment = RoutePlan[Cursor+1]: IDs of previous, and respectively, next segments that for v_i to traverse;

Action : Maneuver Strategy;

Precondition : $((ExperimenterControl() = \mathsf{Go}) \wedge (CrashAvoidance() = \mathsf{Green})) \wedge ((in(CurrentSegment) = \emptyset) \vee (\langle PreviousSegment, \mathsf{Green}\rangle \in TrafficLight(CurrentSegment)))$;
Postcondition : $CrashAvoidance() = \mathsf{Green}$;

begin
  $Candidates \leftarrow \arg\min_{|CurrentLane-\ell|}(\{\ell \in FT_{CurrentSegment}(PreviousSegment, NextSegment)\} \cup \{\infty\})$;
  case $Candidates = \{\infty\}$*:* $ManeuverControl(Stop, CurrentLane)$;
  case $Candidates = \{CurrentLane\}$*:* $ManeuverControl(KeepLane, CurrentLane)$;
  otherwise if $\exists CandidateLane \in Candidates \wedge LaneChangeCrashAvoidance(CurrentLane, CandidateLane) = Green$ **then** $ManeuverControl(ChangeLane, CandidateLane)$;
Function $LaneChangeCrashAvoidance(CurrentLane, TargetLane)$
begin
  if $\{v_j \in V(v_i)$:$CurrentLane_{v_j} \in \{CurrentLane_{v_i}, TargetLane\} \wedge distance(location_{v_j}, location_{v_i}) < SafeDistance\} = \emptyset$ **then return** $Green$;
  `/* Change lane only when no vehicle resides on` v_i`'s trajectory */`
  return Red; `/* current or target lanes, resides behind the safety distance */`

Algorithm 3: Crash avoidance mechanism for vehicle v_i

Input $V(v_i)$: Set of vehicles that are on v_i's line of sight;
Input $location_{v_j}$: Current geographical location of vehicle $v_j \in V(v_i)$ in the Euclidean plain;
Output Red and Green are the two signals, which respectively stand for "no go", and "ok to go";
Local distance(locA, locB): Returns Euclidean distance between locations $locA$ and $locB$;
Constant SafeDistance: Minimum required distance between any two vehicles that are moving in the same lane;
if $\exists v_j \in V(v_i) : OnTrajectory_{v_i}(v_j) \wedge distance(location_{v_i}, location_{v_j}) \leq SafeDistance$ **then return** Red
return Green ; `/* Go only when no vehicle on` v_i`'s trajectory resides behind the safety distance */`

5. Implementation Challenges

Recent advances in the field of mobile robots allow the ad hoc construction of affordable test-beds at your own parking lot. In fact, converting RF miniature vehicles to be a WiFiBot [25] controlled is a popular student project. In this section, we report on our preliminary implementation efforts that uses the Vaillante WiFi [26] as the bases for the miniature vehicle unit.

Vaillante is an off the shelf product. Each remotely controlled miniature vehicle ($1/16$ scale) can go up to 30 Km/h, and can be equipped with several onboard sensors, such as a camera, MicaZ 2.4 GHz motes and localization sensors, e.g., [27]. Each miniature vehicle is remotely controlled by either a human driver or machine (computer program). The human driver controls the vehicle via the drivers interface. Vaillante has its own iPad app interface to which one can consider a video streaming extension by mounting a WiFi camera.

One challenge that the prototype experimenter faces is how to draw the lane markings that the miniature vehicles can drive along. For example, the steering of the Vaillante vehicles does not allow driving along very sharp curves even at their slowest speed. Therefore, we require that every lane marking could be the result of connecting road building blocks.

We considered a set of standard road building blocks. We explain how for the proposed set of building blocks the Vaillante vehicle can follow the lane marking with the aid of mechanisms for lane detection and tracking [19, 20, 23].

In addition, we explore key technological challenges.

5.1 Standard Road Building Blocks

The miniature vehicles follow road markings in order to keep their lanes. All vehicle maneuvers, such as intersection crossing and lane changing, should follow the shortest possible path between the vehicle current and target positions, while keeping safety distance from all other vehicles. Given the fact that we wish to use inexpensive miniature vehicles that move quickly in the test-bed, we have to endure a degree of unpredictability in the vehicle motion. Namely, different miniature vehicles respond differently to the driver's commands and the same miniature vehicle may behave differently under very similar conditions.

Road marking that follows a straight line is rather simple to follow. Therefore, our implementation study mainly considered curves of different angles. Here, the vehicle needs to take several steps before it meets the required angle. Next, we present our single-step experiments

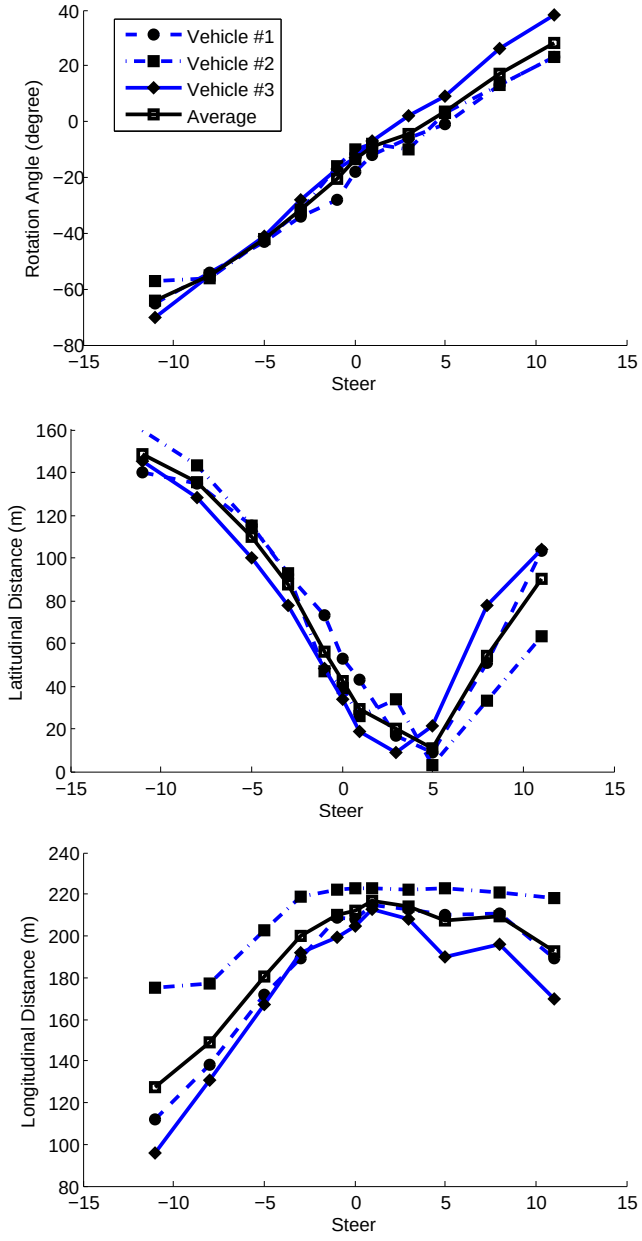


Figure 5.1: Measured behavior of the Vaillante miniature vehicles. The rotation angle as a function of the vehicle steering value is depicted on the top figure. The latitudinal and the longitudinal distances are depicted by the middle, and respectively, the bottom figures.

before discussing the case of step sequences.

We have experimented with the Vaillante miniature vehicles and measured the behavior of three units. All experiments considered a single speed value, 7, and a variety of steering values. For a given steering value, $x \in [-15, 15]$, Figure 5.1 and equation (5.1) estimates the values of the rotation angle, and the distances (latitude and longitude), see Figure 5.2. We have observed

that a single vehicle step is of 197 cm on average and has a divination of 15 cm (when moving on a straight line, the battery is fully-charged, and the steering value is between $+2$ and $+7$, depending on the Vaillante unit).

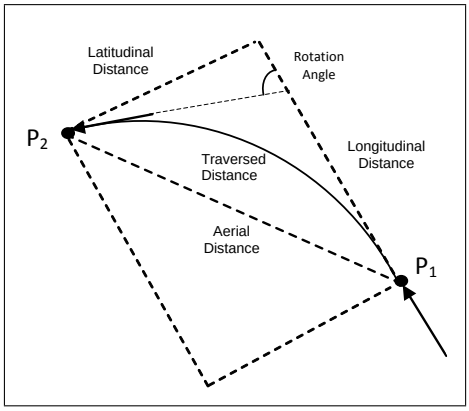
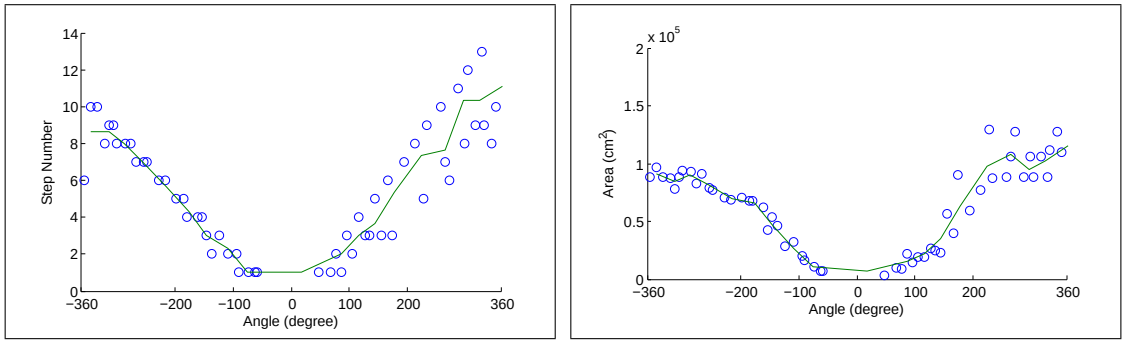


Figure 5.2: Measured vehicle steps. The miniature vehicle is moving on a curve and covers the *traversed distance* from point p_1 to point p_2. Note that the traversed distance is longer than the shortest distance between p_1 and p_2, which is the areal distance. The latitudinal and the longitudinal distances consider a coordinate system in which point p_1 is the origin and the latitude line are perpendicular to the pre-motion vehicle direction, i.e., the vehicle chassis. The vehicle rotation angle refers to the post-motion angle of the vehicle and the longitudinal line.

$$RotationAngle(x) = 0.0003x^4 + 0.003x^3 - 0.053x^2 + 4.956x - 12.991 \quad (5.1)$$
$$Longitude(x) = 0.107x^3 - 4.14x^2 + 42.343x + 86.449$$
$$Latitude(x) = 0.005x^5 - 0.151x^4 + 2x^3 - 12.584x^2 + 10.773x + 148.85$$

Equation (5.1) facilitates the estimation of the vehicle's post-motion position. Therefore, we can estimate the vehicle position after taking a number of steps. Such step sequences facilitate the set of maneuvers, which maneuver controller carries out (cf. $Maneuver()$ in Section 3). Namely, each step takes into consideration the post-motion position of its predecessor.

We use such step sequences when proposing our set of building blocks. This set offers a large variety of rotation angles as presented by Figure 5.3. The figure also suggests boundaries for the rotation angle, as well as the number of steps and size of areas that are required for completing the maneuver. However, as the number of steps grows, so does the deviation from the expected vehicle location. Thus, the $Maneuver()$ primitive must use localization [27] and lane tracking



(a) (b)

Figure 5.3: Number of steps (a) and size of areas (b) that are needed for the completion of simple turning maneuvers

technologies [19, 23] for aligning the vehicle position and adjusting its maneuvers according to the route plan. The idea of the set of building blocks could facilitate drawing both virtual and physical lane markings. A sequence of building blocks makes a feasible lane marking for performing maneuvers by the vehicle. Moreover, the set of building blocks could be used for finding the best maneuver in lane correction and lane changing situations by miniature vehicles.

In order to examine feasibility of the building block set, we implemented two different virtual lane markings, circle lane and eight shape lane. The circle lane facilitates a feasible lane marking simulation and test-bed floor. The eight shape lane provides the simplest form of intersection which the miniature vehicle moves within a loop. Our experiments shows that this model is feasible both in simulation and test-bed floor.

5.2 Estimation of the Miniature Vehicle's Heading

One of the challenges in using vehicles is estimation of the vehicle's heading without using the assistant technologies. We assumed that an accurate localization exists and we know the $position_{v_i}$ (location and heading of the vehicle v_i) before taking the step and v_i's location after taking the step.

Miniature vehicle traverses a curve in each step. This curve can be represented as part of circumference of a circle. Heading of the vehicle at the end of the step is estimated by using slope of the tangent line to the curve in that point. Having two points, start and end point of the step, is not enough to interpolate the circle uniquely. Infinitive number of circles can pass through these two points and heading of the vehicle at the end of the step is varied for each curve. But, if we have the coordination of another point on the curve then we can have a unique curve which passes

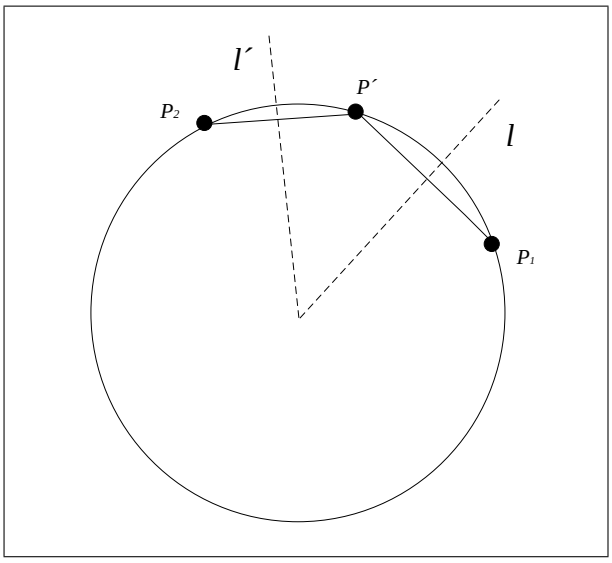


Figure 5.4: Vehicle v_i traverses a step from P_1 to P_2, we can estimate its heading in point P_2

through these three points. Third point can be an intermediate point between start and end point, P_1 and P_2, which we call it $P' = (x', y')$. P''s location can be received from *Base Station* during traversing from P_1 to P_2. Lets assume line l is perpendicular bisector of the line which is passed through P_1 and P', with slope m. Respectively l' is perpendicular bisector of the line which is passed through P_2 and P', with slope m' (see Figure 5.4). Intersection of l and l' is the center of circle (x_c, y_c) which passes through P_1, P_2 and P' points. After finding coordination of the circle's center, it can be used in equation of the circle. Then, heading of the vehicle in point P_2 is calculated by using slope of the tangent line in P_2. The calculation for finding coordination of the center is as following:

$$\begin{aligned} l &: y = m(x - \frac{x_1 + x'}{2}) + \frac{y_1 + y'}{2} \\ l' &: y = m'(x - \frac{x_2 + x'}{2}) + \frac{y_2 + y'}{2} \end{aligned} \tag{5.2}$$

By intersecting lines l and l', we have:

$$\begin{aligned} x_c &= \frac{m(x_1 + x') - m'(x_2 + x') + y_2 - y_1}{2(m - m')} \\ y_c &= m(\frac{m(x_1 + x') - m'(x_2 + x') + y_2 - y_1}{2(m - m')}) + \frac{y_1 + y'}{2} \end{aligned} \tag{5.3}$$

Radius of the circle can be calculated as distance between center of circle and one the points, P_1, P_2 or P' which is called r. The heading of the vehicle can be calculated by using derivation of circle's equation in point P_2:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{5.4}$$
$$Slope\ of\ tangent\ in\ P_2 = \frac{x_c - x_2}{y_2 - y_c}$$
$$Heading\ of\ the\ vehicle\ in\ P_2 = \arctan(\frac{x_c - x_2}{y_2 - y_c})$$

5.3 Safely Following of Virtual Lane Marking with Lane Change Maneuvers

The miniature vehicles follow road markings in order to keep their lanes. All vehicle maneuvers, such as intersection crossing and lane changing, should follow a short path that safely leads from the vehicle's current position to its target. Namely, a safe distance from all other vehicles must be kept, the lane orientation should be kept and, while changing lanes, the departure time should be short. Given the fact that we wish to use inexpensive miniature vehicles that move quickly in the test-bed, we have to endure a degree of unpredictability in the vehicle motion. Namely, different miniature vehicles respond differently to the driver's commands and the same miniature vehicle may behave differently under very similar conditions. Therefore, we design an algorithmic mechanism for following the virtual lane marking and changing lanes. The mechanism merely considers the current position of the miniature vehicle and the virtual lane that it wishes to follow (and does not consider the travel history of the miniature vehicle). Our design criteria satisfy the safety requirements of lane change maneuvers and provide resilience to some degree of unpredictability in the vehicle motion.

We wish to allow the miniature vehicles to perform basic maneuvers, $ManeuverControl(ManeuverCommand, Lane)$, such as keeping their lanes ($KeepLane$), changing their lanes ($ChangeLane$), or halt ($Stop$) as in [23] where full-scale vehicles are considered. The proposed mechanism for lane maintenance with lane changing consider a set of maneuvers that satisfy the safety criteria, see Algorithm 4. It calls $ProposeManeuver$ which selects the optimal maneuver in order to positions the vehicle in $Lane$. $ProposeManeuver$ generates different trajectories that can meet position and curvature constraints. The $ProposeManeuver$ chooses optimum trajectory among aforementioned trajectories by minimizing an utility function that penalizes cross track error, curvature, heading error and departure time [22]. see Figure 3.2 in Section 3.

Algorithm 4: Maneuver Control algorithm, code for miniature vehicle $v_i \in V$

Input $position_{v_i}$: Current geographic location of the v_i, $location_{v_i}$ and its heading;
Input Lane: The lane which the vehicle aims to move to;
Input LaneSuffix($location_{v_i}, Lane$): Returns set of all vertices on the $Lane$, which are positioned ahead of the vehicle v_i;
Local ProposeManeuver($position_{v_i}, LaneSuffix(location_{v_i}, Lane)$): Returns the optimal maneuver from $position_{v_i}$ to the $Lane$ with respect to the heading error, lane cross error and departure time;
Local move($Maneuver$): Sends steering parameters based on the $Maneuver$;
Function $ManeuverControl(ManeuverCommand, Lane)$
begin
  switch $ManeuverCommand$ **do**
    case $ManeuverCommand \in \{ChangeLane\} \cup \{KeepLane\}$:
      $AvailableManeuver =$
      $ProposeManeuver(position_{v_i}, LaneSuffix(location_{v_i}, Lane))$;
      if $AvailableManeuver \neq \perp$ **then** $move(AvailableManeuver)$;
      else $halt$; `/* There is no maneuver in order to reach the Lane */`
    case $ManeuverCommand = Stop : halt$; `/* There is no lane in order to continue the path */`
    otherwise $halt$; `/* error: The input is not valid */`

5.4 Technological Challenges

It is imperative to assure that the miniature vehicles do not crash due to the possible failure of system components, such as (clock) synchronization, communication and localization. Unlike the synchronization, communication and localization requirements in the case of fully deployed vehicular systems, some of the needed autonomic characteristics can be simplified in the case of Gulliver. Therefore, we explain the implementation of these primitives using the coordinating base-station (and consider that as the default implementation) before considering the possible extensions that has autonomous characteristics.

5.4.1 Clock Synchronization

Clock synchronization can simplify the design of the communication primitives and applications for vehicular systems. We base our implementation on the self-stabilizing and autonomous design of Herman and Zhang [28]. Their design is based on the converge-to-the-max technique in which all mobile motes adjust their internal clocks to the maximal time value that they have recently seen.

5.4.2 Communications

Vehicle to vehicle (V2V) communications are carried out via message passing (radio transmissions). Using these messages, the vehicles coordinate their "world" perception and decide about their joint actions. Existing ad hoc communication mechanisms, such as CSMA/CA, can implement V2V communication and facilitate clock synchronization mechanisms [29, 30]. Greater efficiency can be achieved when using a coordinating base station [31]. Recent developments consider media access control with higher predictability [32, 33]. This is another way to go toward an autonomous implementation.

5.4.3 Miniature Vehicle Localization

Positioning systems, such as Global Positioning System (GPS), have a wide range of applications in vehicular systems. Gulliver uses for positioning information in order to inform the vehicles about the locations of the platform entities: vehicles, road segments, intersections, etc.

We consider a design of the default implementation in which each vehicle sends beacons to peripheral receivers (anchors) that have known locations. The anchors then report about the received beacons to a coordinating base station in which a localization algorithm [such as 34] is used for extracting location information before sending this information to the moving miniature vehicles. The above centralistic design allows us to use powerful processing at the coordinating base-station.

6. Virtual Traffic Light

The vehicles moving on roads get signals from traffic light located in each intersection which can be permission to pass or signal to stop. We have designed a *Virtual Traffic Light* (VTL) algorithm which serves the traffic light signaling to the vehicles.

In [17] authors proposed a *Timed Virtual Stationary Automata* programming layer, which provides communication between real clients and virtual nodes layer called Virtual Stationary Automata (VSA). The VSAs are positioned in fixed regions. In our proposed algorithm for *Virtual Traffic Light*, we use this programming abstract. Each intersection segment has a predefined geographic location and different directions (each direction is defined based on the existing lanes in the intersection area). The vehicles communicate with VSAs, each VSA is associated to a direction. The first vehicle entering a VSA region creates a VSA, all vehicles that come in to the region will join to the existing VSA. When a vehicle enter to the segment, sends a join message, if it does not receive a reply, it assumes itself alone in the region, becomes the leader and creates a new VSA. But, if a leader exists in the region, it replies to the vehicle by last state of the VSA.

The programming abstract supports the dynamic environment like a VANET environment which vehicles join and leave the regions continuously. When a vehicle leaves the region, it sends a LEAVE message, and the VSA pops the vehicle's ID from its queue. If the leader leaves the the region, other clients should select a new leader. The clients discover the absence of the leader when they do not receive the periodic messages from the leader. After that, the clients should elect a new VSA leader; one approach can be selecting the client with lowest ID (see [17]).

In our design, the VSA runs VTL which provides the timing for signaling. Vehicles send message to their corresponding VSA, and VSA replies with the last state containing the traffic light state for that direction. See algorithms 5, 6, 7 for proposed design of the *Virtual Traffic Light*.

First version of traffic light in Gulliver is implemented in form of base station traffic light. The traffic light consists of a scheduler which determine the signal ($Green$, Red and $Orange$) for each direction. In intersection design we have considered that there are 16 different directions. The set of directions which can get $Green$ is called $unconflicting\ set$. Base station traffic light broadcasts a vector of all direction states, Red, $Green$ and $Orange$. $Traffic\ Light\ Client$ receives the broadcasted message periodically, and make a decision to $Stop$ or Go. In fact, the $Traffic\,Client$ should retrieve the corresponding field in the vector to its direction. This method is completely centralized and there is no failure resistance, e.g. crashing the base station. So, we implemented next version of the traffic light which uses leader election and one node (leader)

Algorithm 5: Function responsible for communicating the vehicles and their corresponding VSA

Input v_i: Vehicle ID;
Input $position_{v_i}$: Geographical location and heading of the vehicle v_i;
Input $Direction$: Represent the ID of direction which the vehicle v_i aims to enter, sent by *Experiment Manager*;
Output Red, Orange, Green are the three signal, which respectively stand for "emergency stop", "no go", and "ok to go";
Internal event Entering(), Leaving(): Events which are triggered based on position of vehicle toward the intersection, entering or leaving respectively;
External CompareIntersection($position_{v_i}$): A function which checks whether the vehicle is inside one of the intersection areas or not with respect to the current position $position_{v_i}$, if the vehicle is inside the area, function returns True, otherwise it returns False;
External CompareLocation($position_{v_i}$): A function which compares location of the vehicle v_i with geographic info about critical section and returns status of vehicle toward critical section, LEAVE or ENTER;
External event VSA_Update(): A VSA event which returns the VSA state;
Local VSA_Send(message): A function responsible for sending messages to VSA layer;
Local State: The state of the direction which is provided by VSA_Update;
Constant Join, Status, Leave: Message tags which represent type of messages sent to VSA;

Task $Background()$ **begin**
  while $(CompareIntersection(position_{v_i}))$ **do**
    if $(CompareLocation(position_{v_i}) = ENTER)$ **then**
      raise event $Entering()$;
    else if $(CompareLocation(position_{v_i}) = LEAVE)$ **then**
      raise event $Leaving()$;

Upon $Entering()$ **begin**
  $VSA_Send(\langle$Join, $Direction\rangle)$;
  $VSA_Send(\langle$Status, $Direction, ENTER\rangle)$; `/* After sending Status massage, VSA_Update() will be triggered and will return the VSA state */`

Upon $Leaving()$ **begin**
  $VSA_Send(\langle$Status, $Direction, LEAVE\rangle)$;
  $VSA_Send(\langle$Leave$\rangle)$;

Upon $VSA\_Update()$ **begin**
  if *State = Red* **then**
    $VSA_Send(\langle$Status, $Direction, CompareLocation(position_{v_i})\rangle)$;
  else
    return $State$;

Algorithm 6: Function for receiving message by VSA; it's responsible to serve the vehicles which aim to enter the direction

Input v_i: Vehicle ID;
Input $Direction$: Represent the ID of direction which the vehicle v_i aims to enter, sent by *Experiment Manager*;
Local State: The state of the direction which is provided by VSA_Update;
Constant messageType: Type of the received message, $ClientMessage$, $VTLMessage$, $VSAMessage$ which refer to messages from $Client$ (vehicle), VTL and respectively VSA;
Upon $VSA\_Receive(message)$ **begin**
  if $messageType = ClientMessage$ **then**
    $\langle v_i, Direction, Status \rangle \leftarrow message$;
    if $Direction = VSA\_ID$ **then**
      $Send\_VTL(message)$;
  else if $messageType = VTLMessage$ **then**
    $\langle v_i, Direction, State \rangle \leftarrow message$;
    return $State$;

propagates traffic light information. The node with minimum ID becomes the leader. In the startup all nodes consider themselves as leader but they don't broadcast traffic light information. They wait and listen to the other neighbors for a specific time to make sure they have received messages from all other nodes. Since we have assumed that there is no message lost, so all nodes mark the node with minimum ID as a leader. The leader starts to propagate the traffic light state for all directions. If a node with the ID less than the leader's ID join the segment, then all other neighbors consider it as the new leader because they check validity of the leader periodically. This method is more distributed but, in case of leader failure it does not work properly. Assume the leader crashes or leaves the region, the other nodes will not realize it and wait for leader updates. Hence, we implemented third version of virtual traffic light which solves the mentioned problem. In this approach, all nodes send alive messages with their ID and local clock periodically. Moreover, we have used the $converge\ to\ the\ max$ algorithm for synchronizing the nodes in the region.

$TinyOS$ provides timer interface to have the native clock of the MicaZ node. We have defined a local clock which initialize by native clock value. The local clock updates by received local clocks from the neighbors. The nodes send their local clock to all neighbors along with their ID; the neighbor compares the received clock with its native clock, if the received clock is larger than the node's clock then the node's local clock jumps to the received clock value. In other words, the node increases the local clock by difference between its native clock and received clock. Based on this approach, nodes converge to the fastest clock and the approach is called

Algorithm 7: The traffic light VSA function, sets the traffic light state for all directions

Input v_i: Vehicle ID;
Input Direction: The ID of direction that vehicle aims to enter;
Input Status: Sent in Status message by VSA, represents the vehicle status whether it wants to ENTER or LEAVE the critical section;
Input message: The received message;
Output IntersectionState[]: An array that contains state of all directions in the intersection area;
Local DirectionSet[]: Contains information about all legal directions in the intersection area and provided by the Scenario Manager;
Local CarWaiting[]: An array of queues; queue of vehicles which wait for entering to directions, the array index is direction ID;
Local clock: Is a timer;
Internal event $VTL\_Receive(message)$: An event which is raised in receiving messages. The event is triggered by the VSA which makes sure all messages are delivered in same order in all vehicles;
Constant messageType: Type of the received message, $ClientMessage$, $VTLMessage$, $VSAMessage$ which refer to messages from $Client$ (vehicle), VTL and respectively VSA;
Constant TimePeriod: The maximum time which traffic light is Green or Orange;
Upon $VTL\_Receive(message)$ **begin**
  if $messageType = VSAMessage$ **then**
    $\langle v_i, Direction, Status\rangle \leftarrow message$;
    if $(Status = ENTER)$ **then**
      $CarWaiting[Direction].push(v_i)$;
      if $(IntersectionState[Direction] = Green)$ **then**
        if $(clock > TimePeriod)$ **then**
          if $(\exists direction \in DirectionSet \wedge direction \neq Direction \wedge CarWaiting[direction] \neq empty)$ **then**
            $IntersectionState[Direction] \leftarrow Orange$;
          $StartClock()$;
        $VSA\_Send(\langle v_i, Direction, IntersectionState[Direction]\rangle)$;
      else if $(IntersectionState[Direction] = Orange)$ **then**
        if $(clock > TimePeriod)$ **then**
          $IntersectionState[Direction] \leftarrow Red$;
        $VSA\_Send(\langle v_i, Direction, IntersectionState[Direction]\rangle)$;
      else if $(IntersectionState[Direction] = Red)$ **then**
        if $(\forall direction \in DirectionSet \wedge IntersectionState[direction] = Red)$ **then**
          $IntersectionState[Direction] \leftarrow Green$;
          $StartClock()$;
        $VSA\_Send(\langle v_i, Direction, IntersectionState[Direction]\rangle)$;
  else if $(Status = LEAVE)$ **then**
    $CarWaiting[Direction].pop(v_i)$;
    if $(CarWaiting[Direction] = empty)$ **then**
      $IntersectionState[Direction] \leftarrow Orange$;
      $StartClock()$;

$converge\ to\ the\ max$. The nodes keep a list of all alive neighbors and check the list to update the state of the neighbors periodically (neighbors state can be alive or failed). They calculate difference between current value of native clock and the value of native clock in receiving the last alive message from the neighbor, if this value is greater than a predefined threshold, then the neighbor will be marked as failed. By using this method, if the leader fails then the other nodes will realize it after a period and re-elect a new leader. By applying this method, virtual traffic light will be more resilient to the failures.

7. Conclusions

The use of automotive technology can increase traffic throughput by improving vehicle density in roads. Safety requirements can be met by monitoring driver's behaviors without necessarily building new road infrastructures. Thus far, many of the future vehicular systems are not allowed to operate on public roads due to the collision risks. Moreover, the lack of knowledge about the possible emerging patterns in large scale deployment requires additional testing. Gulliver provides an open platform for demonstrating cyber-physical aspects of vehicular systems, making sure that their safety requirements are met and that their emerging patterns assure high traffic throughput.

We presented a design that allows the conduction of a variety of experiments in areas such as vehicle safety (e.g., crash prevention), energy (e.g., multi-lane vehicle platoon), to name a few. This inexpensive test-bed facilitates deployment of testing procedures. For example, protections against vehicle crashing of a miniature vehicle (500 g, 50 cm and 200 to $2k$ Euro) are simpler than a full-scale vehicle (1 ton, 5 m and $100k$ Euro).

Gulliver lies between computer simulation and full-scale vehicle models, and as such, it simplifies and reduces the costs of vehicular system prototyping and development. Note that the cost of each miniature vehicular unit is at least one or two order of magnitude less than a full-scale vehicular prototyping unit. By simplifying prototype development and demonstration processes, Gulliver opens up new research opportunities and promotes cross-fertilization between academic and industrial research.

Nowadays, there are 750 million motor vehicles in the world and the numbers are doubling every 30 years. Vehicular systems will enable vehicular interaction, cooperation and will be the first cyber-physical systems to reach the scale of million units. Currently, no safety-critical system comes close to this scale. Gulliver design is the first to facilitate the detailed investigation of the vehicle interaction and emerging patterns among hundreds and even thousands of units of a cyber-physical system.

BIBLIOGRAPHY

[1] Mitra Pahlavan, Marina Papatriantafilou, and Elad M. Schiller. Gulliver: A testbed for developing, demonstrating and prototyping vehicular systems. In *MobiWac 2011, October 31–November 4, 2011, Miami, Florida, USA*. ACM, 2011.

[2] Mitra Pahlavan, Marina Papatriantafilou, and Elad M. Schiller. Gulliver: A testbed for developing, demonstrating and prototyping vehicular systems. In *VTC 2012, May 6-9, 2012, Yokohama, Japan*. IEEE, 2012.

[3] C.R. Lee, J.W. Kim, J.O. Hallquist, Y. Zhang, and A. Farahani. Validation of a FEA tire model for vehicle dynamic analysis and full vehicle real time proving ground simulations. Technical report, Society of Automotive Engineers, 400 Commonwealth Dr, Warrendale, PA, 15096, USA,, 1997.

[4] Y. Zhang, A. Tang, T. Palmer, and C. Hazard. Virtual Proving Ground-an integrated technology for full vehicle analysis and simulation. *International journal of vehicle design*, 21(4):450–470, 1999.

[5] KM Captain, AB Boghani, and DN Wormley. Analytical tire models for dynamic vehicle simulation. *Vehicle System Dynamics*, 8(1):1–32, 1979.

[6] Michael D. Letherwood and David D. Gunter. Ground vehicle modeling and simulation of military vehicles using high performance computing. *Parallel Computing*, 27(1-2):109–140, 2001.

[7] Jonghyun Kim, Vinay Sridhara, and Stephan Bohacek. Realistic mobility simulation of urban mesh networks. *Ad Hoc Networks*, 7(2):411–430, 2009.

[8] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Goofi: Generic object-oriented fault injection tool. In *DSN*, page 668. IEEE Computer Society, 2003.

[9] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[10] C. Pinart, P. Sanz, I. Lequerica, D. García, I. Barona, and D. Sánchez-Aparisi. DRIVE: a reconfigurable testbed for advanced vehicular services and communications. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, pages 1–8. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[11] Edward A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'08)*, pages 363–369. IEEE Computer Society, 2008.

[12] Philip Levis, Nelson Lee, Matt Welsh, and David E. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In Ian F. Akyildiz, Deborah Estrin, David E. Culler, and Mani B. Srivastava, editors, *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys'03*, pages 126–137. ACM, 2003.

[13] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner. Sumo (simulation of urban mobility). In *Proc. of the 4th Middle East Symposium on Simulation and Modelling*, pages 183–187, 2002.

[14] Ricardo Fernandes, Pedro M. d'Orey, and Michel Ferreira. Divert for realistic simulation of heterogeneous vehicular networks. In *IEEE 7th International Conference on Mobile Adhoc and Sensor Systems, MASS'10*, pages 721–726. IEEE, 2010.

[15] Michel Ferreira, Ricardo Fernandes, Hugo Conceição, Wantanee Viriyasitavat, and Ozan K. Tonguz. Self-organized traffic control. In *Proceedings of the seventh ACM international workshop on VehiculAr InterNETworking*, VANET '10, pages 85–90, New York, NY, USA, 2010. ACM.

[16] Matthew Brown, Seth Gilbert, Nancy Lynch, Calvin Newport, Tina Nolte, and Michael Spindel. The virtual node layer: a programming abstraction for wireless sensor networks. *SIGBED Rev.*, 4:7–12, July 2007.

[17] Shlomi Dolev, Seth Gilbert, Limor Lahiani, Nancy A. Lynch, and Tina Nolte. Timed virtual stationary automata for mobile networks. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS'05*, volume 3974 of *LNCS*, pages 130–145. Springer, 2005.

[18] Michael Spindel. Simulation and evaluation of the reactive virtual node layer. Master's thesis, Computer Science and Artificial Intelligence Laboratory, 2008.

[19] Joel C. McCall and Mohan M. Trivedi. An integrated, robust approach to lane marking detection and lane tracking. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 533 – 537, june 2004.

[20] Yue Wang, Eam Khwang Teoh, and Dinggang Shen. Lane detection and tracking using b-snake. *Image Vision Comput.*, 22(4):269–280, 2004.

[21] O. Amidi and C.E. Thorpe. Integrated mobile robot control. In *Proceedings of SPIE*, volume 1388, page 504, 1991.

[22] Thomas Howard, Ross Knepper, and Alonzo Kelly. Constrained optimization path following of wheeled robots in natural terrain. In Oussama Khatib, Vijay Kumar, and Daniela Rus, editors, *Experimental Robotics*, volume 39 of *Springer Tracts in Advanced Robotics*, pages 343–352. Springer Berlin / Heidelberg, 2008.

[23] Wonshik Chee and Masayoshi Tomizuka. Lane change maneuver of automobiles for the intelligent vehicle and highway system (ivhs). In *American Control Conference*, volume 3, pages 3586 – 3587, June 1994.

[24] Paul Ammann. A safety kernel for traffic light control. Technical Report ISSE-TR-94-06, Department of Information and Software Engineering, George Mason University, 4400 University Drive MS 4A5, Fairfax, VA 22030-4444 USA, 1994.

[25] Robosoft. URL. http://www.robosoft.fr/eng/.

[26] Vaillante. URL. http://sites.google.com/site/controlbytel/.

[27] L. Payá, A. Gil, O. Reinoso, M. Juliá, L. Riera, and L. M. Jiménez. Distributed platform for the control of the WiFiBot robot through Internet. In *Proceedings of the 7th IFAC Symposium on Advances in Control Education (ACE'06)*. Citeseer, 2006.

[28] Ted Herman and Chen Zhang. Best paper: Stabilizing clock synchronization for wireless sensor networks. In Ajoy Kumar Datta and Maria Gradinariu, editors, *Proceedings of Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium (SSS'06)*, volume 4280 of *LNCS*, pages 335–349. Springer, 2006.

[29] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal gradient clock synchronization in dynamic networks. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC'10*, pages 430–439. ACM, 2010.

[30] Philipp Sommer and Roger Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN'09)*, pages 37–48. ACM, 2009.

[31] J. J. Garcia-Luna-Aceves and Chane L. Fullmer. Floor acquisition multiple access (fama) in single-channel wireless networks. *Mobile Networks and Applications (MONET)*, 4:157–174, October 1999.

[32] Pierre Leone, Marina Papatriantafilou, Elad Michael Schiller, and Gongxi Zhu. Chameleon-mac: Adaptive and self-* algorithms for media access control in mobile ad hoc networks. In Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, and Moti Yung, editors, *Proceedings of Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS'10*, volume 6366 of *Lecture Notes in Computer Science*, pages 468–488. Springer, 2010.

[33] Saira Viqar and Jennifer L. Welch. Deterministic collision free communication despite continuous motion. In Shlomi Dolev, editor, *Algorithmic Aspects of Wireless Sensor Networks, 5th International Workshop (ALGOSENSORS'09). Revised Selected Papers.*, volume 5804 of *Lecture Notes in Computer Science*, pages 218–229. Springer, 2009.

[34] Mauricio A. Caceres, Federico Penna, Henk Wymeersch, and Roberto Garello. Hybrid gnss-terrestrial cooperative positioning via distributed belief propagation. In *Proceedings of the Global Communications Conference (GLOBECOM'10)*, pages 1–5. IEEE, 2010.