Hindawi
Journal of Robotics
Volume 2020, Article ID 9345360, 15 pages
https://doi.org/10.1155/2020/9345360

Research Article

Automated Testing of Ultrawideband Positioning for Autonomous Driving

Benjamin Vedder,[1] Bo Joel Svensson,[1] Jonny Vinter,[1] and Magnus Jonsson[2]

[1]*Department of Electronics, RISE Research Institutes of Sweden, Borås, Sweden*
[2]*School of Information Technology, Halmstad University, Halmstad, Sweden*

Correspondence should be addressed to Bo Joel Svensson; joel.svensson@ri.se

Received 9 September 2019; Accepted 14 November 2019; Published 24 January 2020

Academic Editor: Gordon R. Pennock

Autonomous vehicles need accurate and dependable positioning, and these systems need to be tested extensively. We have evaluated positioning based on ultrawideband (UWB) ranging with our self-driving model car using a highly automated approach. Random drivable trajectories were generated, while the UWB position was compared against the Real-Time Kinematic Satellite Navigation (RTK-SN) positioning system which our model car also is equipped with. Fault injection was used to study the fault tolerance of the UWB positioning system. Addressed challenges are automatically generating test cases for real-time hardware, restoring the state between tests, and maintaining safety by preventing collisions. We were able to automatically generate and carry out hundreds of experiments on the model car in real time and rerun them consistently with and without fault injection enabled. Thereby, we demonstrate one novel approach to perform automated testing on complex real-time hardware.

1. Introduction

Accurate positioning is an important technology for autonomous vehicles. A positioning system needs to be both accurate and dependable; thus, there is a need for extensive testing and evaluation. In this paper, we address this need by automating test case generation not only in simulations [1] and Hardware-In-the-Loop (HIL) tests [2, 3], but also on full-scale hardware. To demonstrate this approach, we equip our self-driving model car [4] with an ultrawideband (UWB) positioning system in addition to the Real-Time Kinematic Satellite Navigation (RTK-SN) positioning system it already has and evaluate the performance of the UWB system against the RTK-SN system. Our test method consists of automatically generating random drivable trajectories for our model car, injecting faults into the UWB system and comparing the position outputs of both positioning systems.

The purpose of generating random drivable trajectories is to expose the positioning systems to a wide range of scenarios without having to manually create all the scenarios; instead, some properties on how the scenarios can be created are defined, and tests are generated based on the properties. These properties include the geometry of the area in which the car is allowed to drive, the driving dynamics of the car, and speed limits. The performance metrics of the tests are how well the uwb- and RTK-SN-based positioning systems agree with each other.

To generate the tests, we utilize the Property-Based Testing (PBT) tool ScalaCheck [5]. PBT is an approach to test functional requirements of software [6]. PBT test cases are automatically generated from an abstract model of the System Under Test (SUT), as opposed to being manually written as in the case with unit testing of software.

As we also want to evaluate the fault tolerance of the UWB system, we utilize Fault Injection (FI). The goal of FI is to exercise and evaluate fault handling mechanisms [7]. FI is commonly used across the entire development process of safety-critical systems; from models of hardware [8] and models of software [9] to software deployed and running on the target system [10, 11]. In our case, we use software-implemented FI on software running on the target system, i.e., the positioning system on the model car.

During FI, it is common to run a few different scenarios (inputs) over and over with different faults injected during the runs, while comparing the SUT to the same scenario without faults present, namely, the *golden run*. These scenarios are often created manually, which can be time-consuming when different aspects of a system have to be considered. In our previous work, we have shown how PBT can be used in combination with FI to generate many tests randomly where the golden run can be derived on-the-fly from the model used in the PBT tool [12]. This way functional and nonfunctional requirements can be tested simultaneously using the same test setup, which can reduce the total required testing effort. We have tested this approach on both a simple End-to-End (E2E) system from the AUTOSAR standard [13] and on a more complex quadcopter system simulator [1]. In this work, we extend our approach of performing PBT and FI simultaneously to be used on a system with HIL simulation, as well as on the full hardware.

Further, it is important to have the ability to replay problematic and interesting tests automatically, in order to evaluate if the results are intermittent or repeatable. In case fault injection is performed, the tests often must be repeated with and without fault injection enabled. Therefore, a method to generate a return trajectory that brings the car back to the initial position and orientation from an arbitrary position and orientation on the test track is required. This brings the following challenges:

(i) Instead of simulated time, the system in this study is running in real time. In the context of this paper, real time means that tests are generated part by part as the car is driving. How do we synchronize the test case generation with the system and can we make sure that the PBT tool can keep up with the latency requirements?

(ii) How to reset the state of the SUT between generated tests? Our tests make the model car drive along a random trajectory and hence make it end up in a random position when the test ends. In order to execute the next test, we have to make the model car drive back to the start position from the random position the previous test made it end up in.

(iii) How do we maintain safety while carrying out the tests? We have to avoid generating tests that cause collisions or other dangerous situations.

The SUT in this study is the UWB positioning system mounted on our self-driving model car. The UWB positioning system derives its position estimate by fusing distance measurements to fixed anchors with heading and odometry data from the internal sensors on the model car. Our tests consist of automatically generating trajectories for the model car with a geometry such that they can be followed by the car, in addition to having the property of not leading the model car into a corner where it has too little space to stop safely. As the car follows the trajectories, a deviation between the UWB and RTK-SN positions exceeding 1 m is considered a failure. During the tests, we also inject faults into the UWB positioning system to study their effect and how they are handled. We perform the experiments both as a HIL test with the main controller of our model car running a simulation of the motor and dynamics of the car and with the model car running outdoors in real time carrying out the auto-generated test cases.

This experiment setup has the real-time and latency challenge and especially the challenge of resetting the SUT state so that new experiments can be performed. Resetting the SUT state here means generating a trajectory to accurately drive the model car back to the initial position and resetting the state of the UWB positioning system. Being able to reset the state consistently is also important for replaying and analyzing recorded experiments. Further, it is important to generate tests that do not make the model car collide with any obstacles, which is a significant challenge compared to the simulated and the HIL cases. Thus, the main contributions of this paper are

(i) Showing how PBT with FI can be carried out on real-time hardware while addressing the aforementioned challenges.

(ii) Our method to generate safe and random trajectories for the model car and how to generate a trajectory to drive the car back to the initial position between tests.

(iii) Showing how to repeatedly replay experiments on real-time hardware with and without FI to study the effects of faults and the effects of random variations in the test environment.

(iv) A method for doing FI in the firmware running on the final hardware with small intrusion on the code base.

The remainder of this paper is organized as follows: In Section 2, we describe the different parts of our testing setup and SUT, and in Section 3, we describe our approach for test case generation. Section 4 presents the results from our tests, and in Section 5, we present our conclusions from this work.

2. System Setup

The SUT in the experiments is the UWB positioning system on our self-driving model car, which is enabled by connecting an UWB ranging module to the embedded controller of the car. A photo of the model car with the UWB module mounted on a pole at the back is shown in Figure 1. Figure 2 shows a block diagram of the model car, where the UWB module is shown at the top in green, connected over a Controller Area Network (CAN) bus. The model car estimates its position by combining RTK-SN [14] with dead reckoning based on the Inertial Measurement Unit (IMU) and odometry feedback from the motor controller (https://vesc-project.com/). The position filter of the car also keeps track of the time stamps from the RTK-SN samples to compensate for the latency of the samples at higher speeds. This makes it possible to estimate the position of the model car with an accuracy of around 5 cm with 100 Hz update rate under dynamic conditions [4].

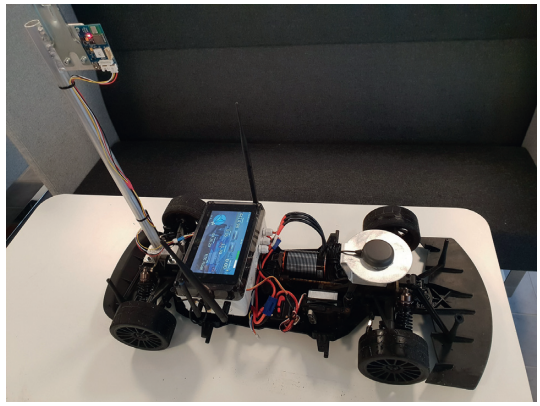

Figure 1: Photo of our self-driving model car with our UWB module attached to the back on a stick.

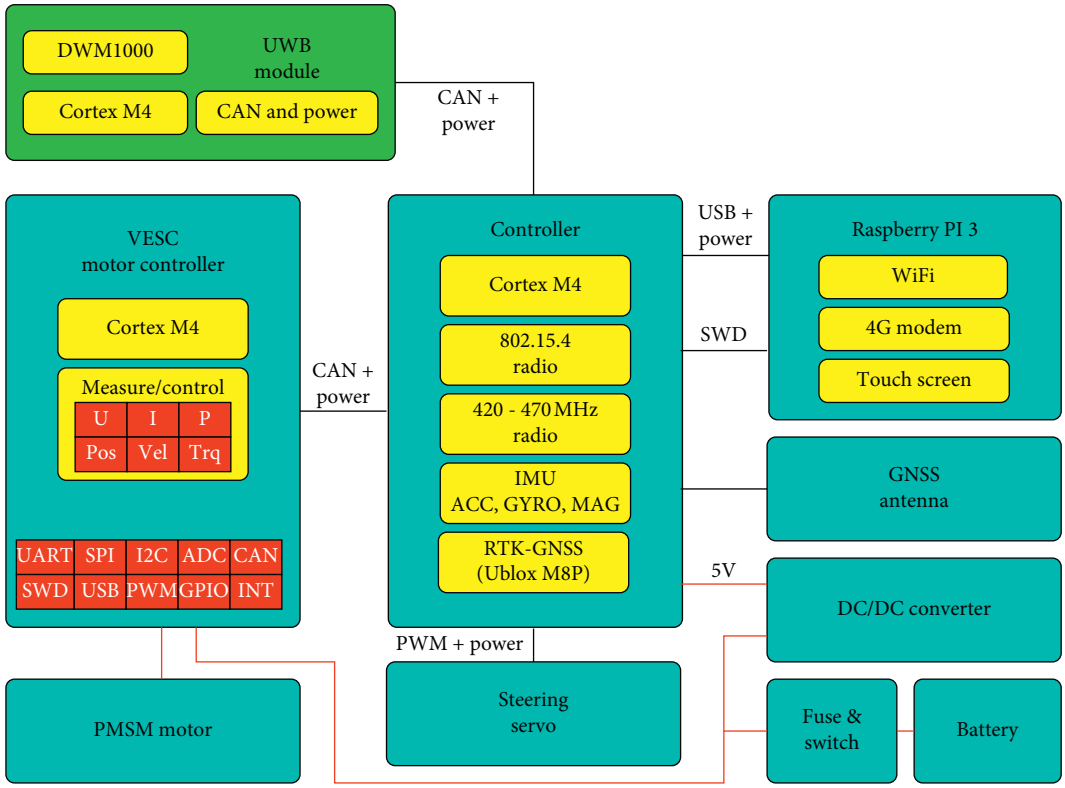

Figure 2: Block diagram of our self-driving model car with our UWB module in green.

To control the model car, we use an SSH tunnel over a 4G connection to the Raspberry PI 3 single-board computer on it, forwarding the ports necessary to control and visualize the state of the model car. The configuration, control, and visualization of the model car is handled from our RControlStation software that runs on a computer to which the SSH port forwarding from the car is done. RControlStation can be used to graphically edit trajectories overlayed on OpenStreetMap [15], which the car can follow using an improved version [4] of the pure pursuit algorithm [16]. The real-time position estimation and control, including the pure pursuit algorithm, is handled on the controller board on the car with a Cortex M4 microcontroller. Remote debugging and firmware updates can be done over the SSH tunnel to the raspberry pi computer, which is a convenience when developing and testing in general.

RControlStation also has a network interface that can be used to control the model car from remote software by sending XML messages over UDP or TCP. Additionally, there is a synchronous C library for generating and decoding these messages, making it easy to implement communication with the cars from any programming language that supports a native interface to C code. We have extended this network interface and library with support for accessing trajectories on the map in RControlStation, as well as support for uploading generated trajectories to the map and/or the model car. This C interface together with the BridJ (https://github.com/nativelibs4java/BridJ) native interface

for Java and Scala gave us full-control over the model cars as well as a visualization interface from Scala. An overview of the setup is shown in Figure 3.

2.1. HIL Simulation Mode. The controller of our model car also has a simulation mode, where the motor and inertia of the car are simulated and its position is updated by feedback from the simulated motor. The IMU and RTK-SN correction is switched off in this mode. The simulation mode enables us to only connect the controller Printed Circuit Board (PCB) of our model car to RControlStation over USB and integrates seamlessly with the rest of the test setup. This is useful for designing and setting up experiments without risking damage to the hardware as dry-runs can be made with most of the software running on the final hardware without physical moving parts.

2.2. UWB Positioning. Our UWB positioning system is responsible for estimating the position of the model car without relying on RTK-SN positions, thus providing an independent and redundant position estimate. This can be useful in scenarios where not the whole driving area has Global Navigation Satellite System (GNSS) coverage or when an independent position source is required to increase the integrity of the position estimate.

The UWB positioning system consists of a number of modules containing a Decawave DWM1000 UWB transceiver [17], a 32-bit microcontroller, a CAN-bus interface, and some voltage regulators. We have developed these modules in previous work [18], and they can measure the distance between each other with best case accuracy of around 10 cm. One of the modules is mounted on the stick on the back of our model car and connected to its controller over CAN-bus, as shown in Figures 1 and 4, and two or more modules are mounted on stationary tripods as anchors shown in Figure 5.

We have extended the firmware of our model car controller with a position sensor fusion algorithm that merges distance measurements from the UWB module on the car to the anchors with odometry data from the motor controller and heading information from the IMU. Further, we have extended RControlStation with the ability to edit the UWB anchor positions on OpenStreetMap and upload them to the model car to be used in the sensor fusion. This gives the UWB sensor fusion the following input information:

(1) The measured distance between the UWB module on the model car to the anchors with known positions.

(2) Odometry, showing how far the car has traveled between iterations of the filter, by means of how much the motor has rotated together with the known gearing and wheel diameter.

(3) Heading information from the IMU derived from the gyroscope and/or the magnetometer, depending on the car configuration.

The heading and odometry information arrives at 100 Hz, while the UWB module is sampled at 10 Hz. Note

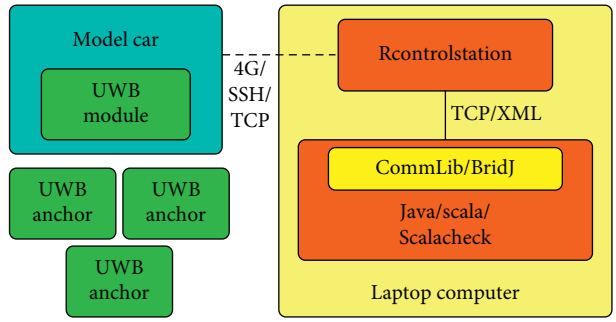

Figure 3: Experimental setup consisting of a laptop computer, our model car, and two or more of our UWB anchors.

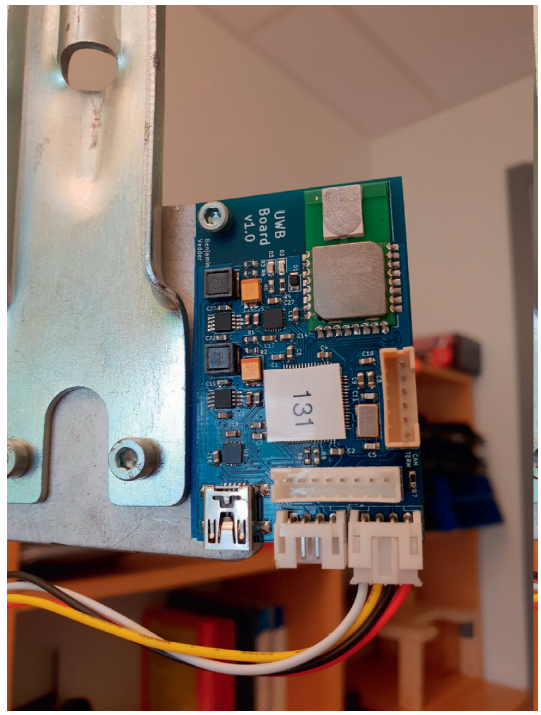

Figure 4: Our custom UWB module mounted on the model car and connected over CAN-bus. Power is also provided through the same connector.

Figure 5: Our UWB modules mounted on tripods, each in a waterproof enclosure together with a battery. These are used as stationary anchors.

that the distance measurements to the anchors are taken one at a time by cycling through a list with anchors provided by RControlStation and only taking measurements of the anchors that are closer than 80 m away from the last position estimate. The 80 m cut is done because the measurements of anchors too far away are likely to not succeed, which would just lower the system update rate.

Based on the odometry and heading data, the position estimate of the car is advanced as

$$p_{xy} = p_{xy\_\text{old}} + D_{\text{tr}} \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}, \tag{1}$$

where p_{xy} is the new position, p_{xy_old} is the previous position, D_{tr} is the odometry travel distance since the previous position update, and α is the heading angle from the IMU. This update is done at 100 Hz and is accurate over short distances, but the error increases without an upper bound as it is based on relative measurements only. To deal with the drift, the 10 Hz distance measurements to the anchors are used one at a time as they arrive to correct the position estimate. This is done by first calculating the expected position of the UWB module on the car by removing its offset from the estimated car position as

$$p_{uwb} = p_{xy} + O_{xy}^{T} \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}, \tag{2}$$

where p_{uwb} is the estimated position of the UWB module on the car, p_{xy} is the estimated position of the car, and O_{xy}^{T} is the offset from the UWB module to the center between the rear wheels of the car and α is the heading angle. Then the estimated distance and direction relative to the achor in question are calculated as

$$\begin{aligned} d_e &= \left\| a_{xy} - p_{uwb} \right\|, \\ v_{uwb} &= \frac{\left(a_{xy} - p_{uwb} \right)}{d_e}, \end{aligned} \tag{3}$$

where d_e is the estimated scalar distance to the anchor, a_{xy} is the position of the anchor, and v_{uwb} is a vector with length l pointing in the estimated direction of the anchor. Then the estimated position of the car is updated as

$$\begin{aligned} c &= \begin{cases} d_e - d_m, & \text{if } \left| d_e - d_m \right| < 0.2 \text{ m}, \\ 0.2 \dfrac{\left| d - d_m \right|}{d - d_m}, & \text{otherwise}, \end{cases} \\ p_{xy} &= p_{xy\_\text{old}} + c v_{uwb}, \end{aligned} \tag{4}$$

where p_{xy} is the new position estimate, p_{xy_old} is the old position estimate, and d_m is distance measured to the anchor by the UWB module.

In other words, when a distance measurement of an anchor done, the measurement is compared to the expected distance of that anchor from the current position estimate and the position estimate is then updated by moving it in the direction of the anchor. This movement is truncated to 0.2 m in order to reject high-frequency noise and outliers such as reflections. This converges well when the initial position is known and at least two anchors are used. If the initial position is unknown and the car is stationary, at least three anchors with positions that are linearly independent in the xy-plane are required for the position to converge. Experiment results on the performance of this position estimation implementation can be found in Section 4. It should be noted that our UWB positioning implementation assumes that the model car moves on a plane and that the anchors have the same height as the UWB module on the car. Deviations from this assumption degrade the UWB positioning performance, but in our experience, the practical impact on the performance is less than 0.5 m, which is within our requirements.

3. Test-Case Generation

For automating the generation of test-cases and the testing itself, we use ScalaCheck [5], which is a framework for testing, primarily, Java and Scala programs implemented in the object-oriented and functional programming language Scala. Beyond this purpose, via BridJ and a C Application Programming Interface (API), to run tests against an embedded system in real time while it is moving around in the physical world.

ScalaCheck provides a library for testing stateful systems based on a sequence of interactions via an API. This library is called the "Commands" library (org.scalacheck.commands.commands), where each possible interaction with the API of the SUT is described as a command that ScalaCheck can generate using a "generator." The Commands library is useful when parts of the SUT state have to be known for generating further commands or when the state is required to determine whether commands produce correct results. When using the Commands library, an abstract model of the SUT state is carried along the test, and each command can use the state to determine if the results it produces are correct. The commands are also responsible for updating the state if necessary. The generator is responsible for generating a suitable command with suitable parameters based on the current state. What the state that ScalaCheck maintains for us looks like is shown later, after introducing all necessary concepts.

The API for interacting with the SUT that we give to ScalaCheck is defined as follows (Algorithm 1).

Ultimately these methods communicate with the model car, as it is running, over TCP. As indicated by the comment provided in the API description, the *runSegments* method can return false when the UWB- and RTK-SN-derived positions differ by over 1 m. If this happens, the test will fail.

When using the Commands library, a ScalaCheck test is a sequence of random commands acting on the system. The commands we have specified are as follows:

(i) RunSegment is a command that carries a generated random trajectory.

(ii) AddFault is a command to inject a fault of some kind.

```
(1) class Car {
(2)   //Apply brake and wait until the car has stopped
(3)   def stopCar(): Unit = {...}
(4)   //Send the next trajectory part to the car and wait until
(5)   //it is almost finished. Returns true if the maximum difference
(6)   //between the UWB and RTK-SN position stayed below 1 m
(7)   def runSegments(route: List[RpPoint]): Boolean = {...}
(8)   //Drive the car back to its initial position and wait until it
(9)   //arrives. Returns true if successful, false otherwise.
(10)  def runRecoveryRoute(ri: RouteInfo, carRoute: Int): Boolean = {...}
(11)  //Add a fault to one of the probes
(12)  def addFault(probe: String, faultType: String,
(13)    param: Double, start: Int, duration: Int): Unit = {...}
(14)  //Clear all faults
(15)  def clearFaults(): Unit = {...}
(16)  //Set the UWB position equal to the RTK-SN position
(17)  //in order to start from a known state.
(18)  def resetUwbPosNow(): Unit = {...}
(19) }
```

Algorithm 1

Each command can be generated using different sets of parameters. In the case of the *RunSegment*, the parameter is a random trajectory while *AddFault* is parameterised on the type and magnitude (or value) of the fault to inject. The following generators of commands are explored in this paper.

(i) *genRunSegments* generates a random trajectory command. The generation procedure is outlined in Section 3.2.

(ii) *genFaultWheelSlip* generates an *AddFault* command carrying information how to alter the car travel distance to simulate this fault. This is explained further in Section 3.1.

(iii) *genFaultYaw* generates an *AddFault* command, with parameters that influence the estimated yaw angle of the vehicle.

(iv) *genFaultAnchor* generates an *AddFault* command, with parameters that add an offset to the measured distance from the anchors.

Each command comes with a precondition and a postcondition. The precondition is used by ScalaCheck to decide if the command can be allowed to run given the current state. After executing the command, ScalaCheck runs the postcondition code and decides if the test can continue or if a failure occurred.

As our tests involve moving a car around outdoors, a parameter to the test-case generation is a description of the area it is allowed to travel in. The geometry of this test scenario is specified using RControlStation by means of routes with different IDs. Note that routes in RControlStation consist of a connected set of points, and we use them for defining both polygons and trajectories.

Figure 6 shows such a scenario where route 0 defines the trajectory that the car starts with in every test and route 1 defines the *recovery trajectory*, which is used to lead the car back to the start trajectory with a repeatable heading and speed. Route 2 defines an outer polygon that encloses where the car is allowed to drive, and routes 4 to 7 define cutouts in that enclosing polygon that the car is not allowed to drive in.

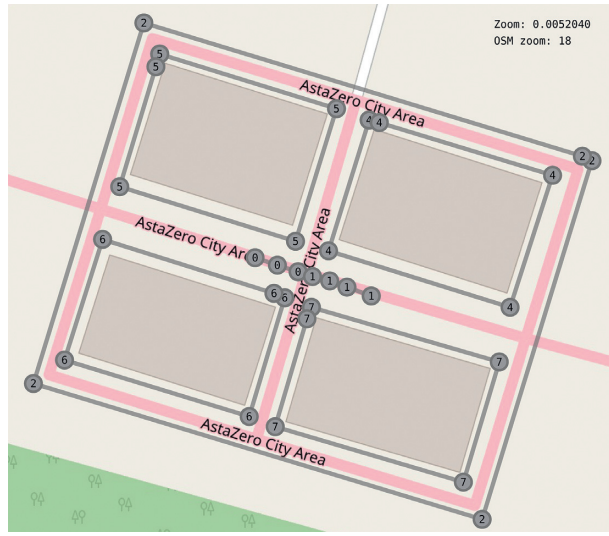

Figure 6: A driving area defined by route with ID 2 with the cutouts 4, 5, 6, and 7. Route 1 is the recovery route and route 0 is the start trajectory. The route IDs are written in the vertex circles of the polygons.

With these concepts established, the state maintained by ScalaCheck to facilitate test-case generation has the following content (Algorithm 2).

That is, it contains the route generated so far as well as information about the test scenario geometry and the number of faults injected so far.

Now we proceed to generate test cases, using ScalaCheck, as follows:

```
(1) case class State(
(2)   //The trajectory that has been generated so far
(3)   route: List[RpPoint],
(4)   //Object containing the drivable map with methods
(5)   //to generate valid trajectories within it.
(6)   routeInfo: RouteInfo,
(7)   //Number of faults injected so far
(8)   faultNum : Int)
```

Algorithm 2

(1) Create a new state and initialize it to default values. The member *faultNum* is initialized to 0 and the member *route* is initialized to the route with ID 0 fetched from RControlStation over TCP/XML. The member *routeInfo* is initialized with the routes 2 and 4 to 7, defining the valid area the experiments are allowed to be generated within.

(2) A new SUT object is created, representing the connection to the car and the actions it can execute. The SUT object is then used to clear the injected faults on the car, drive back to the recovery route, as described in Section 3.3, and to start driving along the start route. Between finishing the recovery route and driving along the start route, the UWB position estimate on the car is reset to be equal to the RTK-SN position estimate. This is done so that all experiments are started in a known and repeatable state.

(3) Now that the state is initialized and the car drives along its start route, a new command is generated, which can be either one of the FI commands, as described in Section 3.1, or a *RunSegments* command, as described in Section 3.2. The commands are generated according to a distribution of choice and can be based on the system state, for example, such that only a certain number of FIs are generated in each test. The *genRunSegments* generator requires the previous trajectory to make a valid extension to it, and the *RunSegments* command needs to update the state with the now extended trajectory.

(4) Repeat (3) until the test is finished or until the post-condition of any command fails. The test size, or number of commands to be generated, can be passed to ScalaCheck when starting the test generation. We have chosen to generate tests with 5 to 20 commands. When running many tests, ScalaCheck will start by generating smaller tests (fewer commands) and increase the test size towards the end of the testing campaign.

(5) After the test is finished, ScalaCheck will call *destroySut*, which in our case tries to stop the car safely. This is done by first generating a valid trajectory with a low speed connected to the current trajectory, waiting until the car has reached the low speed along that trajectory and then applying brake until the car has stopped. This safe stop addresses Challenge III from Section 1.

(6) ScalaCheck will run steps (1) to (5) for the number of tests we decide to run. We usually run between 3 and 100 generated tests like this, depending on the available time and remaining battery life of the model car.

Note that the connection to the model car has a certain latency, we have a limited state sampling interval, and that generating trajectories takes a certain time due to the complexity and potentially large number of tries, as explained in Section 3.2 and 3.3. Therefore, the RunSegments command will not wait until the car has reached the end of the segment, but only until it has a certain time left before reaching the end. During this time, we have to send possible FI commands and the next RunSegments command. As a consequence, we also have to make sure that each RunSegments command provides at least a long enough trajectory to account for this time and that our code is optimized to a certain extent. The fact that ScalaCheck runs on the efficient Java JVM has made it easier to write complex algorithms that execute in a short time. This addresses Challenge I and III from Section 1.

It should also be noted that *shrinking* is a common concept within PBT [6], meaning that failing test cases are shrunk to smaller failing test cases to make analyzing them easier. With the Commands implementation of ScalaCheck, shrinking would mean to remove commands from the failing command sequence while keeping it valid and rerun it after each shrinking step until the shortest sequence of command leading to a failure is found. In our tests, this was not meaningful as changing the command sequence has a significant impact on the experiment and often leads to finding a different fault that happens to result from a shorted sequence of the initial commands.

As our test setup is rather complex to replicate with many details involved, we have published the complete source code for the test generation and all parts for our model car, as well as the sources for RControlStation, on Github under the GNU GPL version 3 license. See the footnotes below for links (https://github.com/vedderb/rise_sdvp, https://github.com/vedderb/rise_sdvp/tree/master/Linux/scala_test, https://github.com/vedderb/rise_sdvp/tree/master/Linux/RControlStation).

Compared to other HIL testing setups, our approach has a more general description of the test scenarios. For example, it is common to choose from a set of traffic scenarios [19] or mission profiles [20] that have to be constructed manually, whereas we completely generate the scenarios based on higher level geometric constraints. This allows us to generate a wide variation of scenarios with little manual work, but brings the challenge of added complexity to the test case generation. Further, our generated tests are not only run in a HIL setup; they are also executed on the full hardware with the additional challenges of keeping the tests safe and being able to restore the state so that tests can be rerun or further tests can be executed.

3.1. Fault Injection. We have based the FI on the approach used by the FaultCheck tool that we have developed in previous work [1, 12]. Essentially, this is done by adding

probes to variables in the firmware of the controller on the model car and controlling these probes with a simplified embedded C version of FaultCheck. For example, we have added probes to the travel distance and yaw variables in the IMU and odometry update described in Section 2.2 as (Algorithm 3).

These probes are controlled by our embedded C FaultCheck library, which is controlled from ScalaCheck using command generators such as (Algorithm 4).

yielding an *AddFault* command. In this example, we generate a wheel slip fault that can be modeled by an amplification greater than one of the travel distances measured by the odometry. For adding faults, the probe has to be specified, the type of fault (BITFLIP, OFFSET, AMPLIFICATION, or SET_TO), the start iteration and number of iterations of the fault. As with FaultCheck, multiple faults can be added to the same probe (or variable). Our C version of FaultChec consists of less than 400 lines of code, has low runtime overhead, and is written without the requirement for external libraries. To use it, only the probes as shown above have to be added, and text strings controlling the faults have to be provided. These text strings can be easily generated by ScalaCheck and sent over the existing communication interface to the model car. In summary, this is a simple method of adding FI support to the firmware with small intrusion on the code base.

In addition to the wheel slip fault shown above, we inject the following faults:

(i) Ranging reflections, meaning that the distance measured between the UWB module on the car and an anchor was not the line of sight, but a reflection. This fault can appear, e.g., when something is blocking the way. We model this by a positive offset fault injected on the measured distance.

(ii) Yaw error, meaning that the yaw angle used for the position estimation as described in Section 2.2 has an offset, which can be caused by, e.g., external objects interfering with the magnetometer. This can be modeled by a positive or negative offset added to the yaw angle.

There are other techniques to inject faults on embedded target hardware, such as scan chain implemented FI [10] that have no intrusion on the final firmware. However, they would require additional code and hardware for controlling the debug port of the controller of our model car from ScalaCheck. Using these techniques also makes it more difficult for time injections to align with variable updates from the external events. Therefore, we considered the small intrusion on the source code a better option in our case given the simplicity and exact control over timing in relation to external events.

3.2. Trajectory Generation. One of the essential parts of our test case generation is the ability to generate random trajectories. Trajectory generation is a known problem within mobile robotics, and it is common to solve problems such as finding parking spots while avoiding obstacles [21] or navigating to a position on a map while adjusting the trajectory around obstacles [22]. Our situation has some similarities with these problems, but our problem formulation is different: we are not aiming for a specific final position or orientation of our model car, we want to generate long random trajectories that are drivable by our model car while staying within the valid driving area.

The trajectories we generate have to stay within the valid outer polygon of the map without crossing the inner polygons, and they must have a shape that our model car can follow given its steering geometry. Our trajectories consist of points creating segments with a length between 0.6 and 2.0 m. The angle between two consecutive segments must be less than 30°, because that makes the tightest turn we can make larger than the minimum turning radius of our model car. Figure 7 illustrates how we generate such valid trajectories randomly.

(1) Assume that we start with a valid trajectory segment, such as the start trajectory 0 in Figure 6. If we start from the car, we make a short segment with the same orientation as the car.

(2) From the previous segment, S_{n-1} in Figure 7, extend three lines of length L; one pointing in the same direction as the segment (line B) and two lines pointing $\pm\,\theta^{\circ}$ to the sides (line A and C), where $\theta = 30^{\circ}$ and $L = 2$ m in our case. If the lines intersect with any of the polygons, truncate them at the intersections (I_1 and I_2).

(3) Create a horizontal rectangle that contains the vertices of the lines A to C, by simply setting the X and Y coordinates to the respective maximum and minimum X and Y coordinates of the line vertices. If the diagonal of this rectangle is less than 0.6 m, we assume that we are stuck in a corner and start over from the start trajectory in Figure 6.

(4) Generate a random XY-coordinate within the rectangle and consider the line segment formed by that coordinate and the end coordinate of segment S_{n-1}; if the segment is between 0.6 and 2.0 m long, it does not intersect with any of the polygons and has an angle of less than 30° to S_{n-1}; we keep this segment and proceed to the next step. Otherwise, we repeat this step until we either generate a valid segment or have reached the maximum number of inner tries in which case we start over with the trajectory generation.

(5) If (4) creates a valid point, we add it to the trajectory as S_n and start over with step (1) with the now extended trajectory. Repeat step (1) to (4) until we have generated the desired amount of points or until we have exceeded the maximum number of *outer* tries.

In our case, we use up to 50 inner tries generating a point within the rectangle for each segment and up to 5000 outer tries of starting over. This way we can successfully generate trajectories of 30–40 segments most of the time, with few retries in the normal case (<2 inner tries and <3 outer tries). This becomes increasingly difficult though when generating trajectories longer than 40 points as it becomes increasingly

```
(1) void pos_uwb_update_dr(float imu_yaw, float turn_rad, float speed) {
(2)     fi_inject_fault_float("uwb_travel_dist," &travel_dist);
(3)     fi_inject_fault_float("uwb_yaw," &imu_yaw);
(4)     ...
(5) }
```

Algorithm 3

```
(1) def genFaultWheelSlip(state: State): Gen[AddFault] = for {
(2)     param ⟵ Gen.choose(10, 50)
(3)     start ⟵ Gen.choose(0, 100)
(4)     duration ⟵ Gen.choose(1, 10)
(5) } yield AddFault("uwb_travel_dist," "AMPLIFICATION," param.toDouble/10.0, start, duration)
```

Algorithm 4

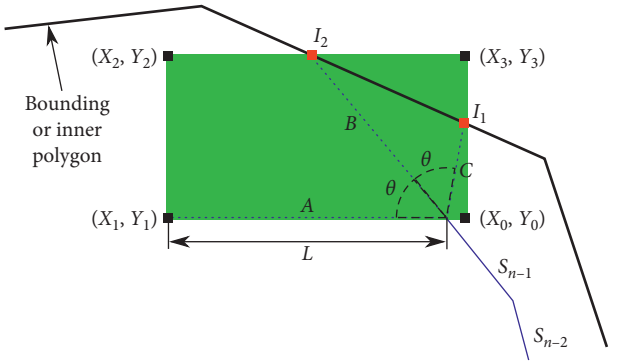

Figure 7: Rectangle considered for trajectory segment generation.

likely that we end up in a corner or against a wall and have to start over. We solve this by generating shorter trajectories with an *ahead margin* and concatenate them. That means that we generate for example 30 points and only use the 10 first points and append them to our trajectory, e.g., with an ahead margin of 20. Thereby, we guarantee that there exists a trajectory of at least 20 points that can be appended to our current trajectory. Then we repeat this for every extension of our trajectory, where the extensions guarantee that there is a possible further extension after it. The size of the ahead margin presents a trade-off between computational load and likelyhood to end up in a corner. The ahead margin of size 20 was found to work well as a result of trial and error. With this method in place, we were able to generate trajectories millions of points long without any issues, even within complex polygons.

This is the method we use to generate a trajectory for the *RunSegmens* ScalaCheck command described above; we generate between 2 and 20 points at a time and guarantee that there exists a trajectory of at least 20 (ahead margin) points with 0.6 to 2.0 m spacing after this trajectory piece. Even in the worst case where we have 5000 outer retries, this generation takes less than 50 ms on a common laptop computer meaning that we can do it in real time, addressing Challenge I.

Our trajectory generation method is not guaranteed to succeed, meaning that we possibly can generate a trajectory leading into a corner with too short distance to stop the car safely. To avoid that, we have a method in our test suite that attempts to generate many long random trajectories within the test area that we use every time we create a new test area. If we are able to generate long trajectories, e.g., 100 km, without exceeding half the maximum number of outer tries we consider it safe to use the given test area, addressing Challenge III. Figure 8 shows an example of a 14 km long generated trajectory within the driving area from Figure 6. The reason that we stopped at 14 km for making the figure is that rendering the trajectory during generation is resource-intensive and not necessary for only making the test. An observation that gives us further confidence in our trajectory generation method is that the only problematic test areas we have found so far have a narrow path longer than the ahead margin leading into a corner with too little space to turn around the car, which is evident by just looking at the test area. Even these areas can be handled though by increasing the ahead margin at the expense of computational power.

3.3. Return Trajectory Generation. Initially, or after finishing previous experiments, the model car is located at a random position with a random orientation. To minimize the need to manually reorient and move the model car while performing tests, we designed a method for automatically generating a return-route. The goal is to connect a starting point and orientation (point-orientation) with a goal point-orientation derived from the beginning of the recovery route while adhering to the same constraints for trajectory generation as the test-trajectory generation, i.e., the trajectory cannot leave the test area, the trajectory may not enter into cut-off areas, and it must be possible for the model car to follow the trajectory given the vehicle dynamics. The method shown in this section addresses Challenge II from Section 1.

Our method of return-route generation is based on the following insights or heuristics:

Figure 8: Part of a 14 km long trajectory generated within the driving area from Figure 6.

(1) If unconstrained by obstacles, an efficient way to reposition and reorient a car is to turn in an arc, using maximum steering angle, towards the new position while making room for the turning arc necessary to also establish correct orientation at the target position. Figure 9 illustrates this approach to repositioning the model car.

(2) From Section 3.2, we already have a method for generating random trajectories that can reach most locations within the test area. So if it is impossible to reposition the car using (1), a short random trajectory can be extended from the current position leaving us in a new location to try again from. This procedure of trying (1) and (2) is iterated for a maximum number of tries before giving up and restarting from the original point-orientation. After finding a valid trajectory, we start over with this process again until we find a new valid return trajectory and keep it if it is shorter than the previous one. This is then repeated 100 times in order to increase the probability of generating a shorter return trajectory.

(3) From a number of repetitions of trying and retrying (1) and (2), we are likely, but not guaranteed, to have a route that connects the starting point-orientation with the target point-orientation. Given the random nature of this approach, we are however not likely to have obtained the shortest route (or even a short route by any standards). We improve on this by applying a trajectory shortening pass to the generated trajectory. This pass is explained in more detail below, but, in essence, it attempts to find valid shortcuts between head and tail sublists of the generated trajectory. While this methodology still does not guarantee that the route is optimal, we have seen that in practice the result is an improvement. As an example, Figure 10 shows a trajectory generated by our approach without optimization, and Figure 11 shows the same trajectory after the optimization pass. In this case, the trajectory was shortened from 104.4 m to 84.9 m.

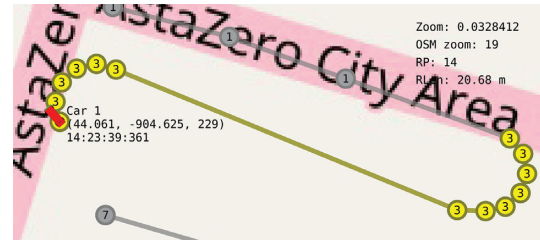

Figure 9: Trajectory generated by extending arcs from a starting point-orientation (the car) to a goal point-orientation (the recovery trajectory (trajectory 1)).

Figure 10: Recovery trajectory generated without optimization, length: 104.4 m.

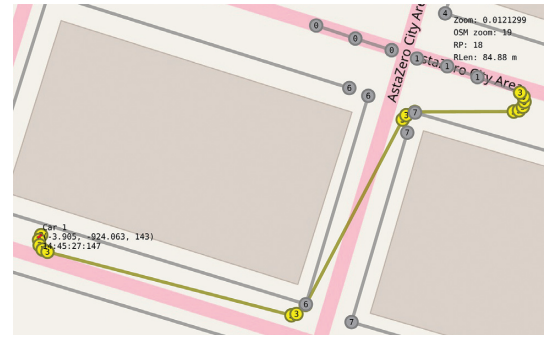

Figure 11: Trajectory from Figure 10 after the optimization pass, new length: 84.9 m.

More in depth, the method for connecting two arbitrary point-orientations by arcs and a line is performed as follows: *First*, extend arcs, consisting of short line segments, turning left and rightwards from both the start and target positions. *Second*, try to connect initial subarcs at the start and target position by a line given the constraint that it must have a valid turning angle. If it is possible to connect the start and target position and the resulting trajectory is valid the procedure ends successfully; otherwise, failure to find a direct route is reported.

As Section 3.2 explains step (2) in depth, we now proceed to look at the details of (3), trajectory shortening. Our method expects a trajectory as input and tries to shorten it

while maintaining the start and end positions and orientations intact, as well as respecting the constraints of a maximum turning angle between consecutive segments and not intersecting with the polygons. The shortening process iterates over the trajectory in each point cutting it into a head and tail portion. The head portion of the trajectory is extended with arcs turning left and right. These added arcs are then traversed, and in each point, an attempt is made to form a trajectory that meets a tailing sublist of the tail section of the trajectory. If the steps above result in a shorter trajectory, the procedure is run again with this new shorter trajectory as input. When no further shortening can be obtained, the process completes.

3.4. Timing Considerations. As the tests are generated while the model car is driving, the generation has to be fast enough for the car not to reach the end of the trajectory before the next part is generated.

The generation of the next trajectory part is triggered when then car has less than n segments left of the current trajectory. In the worst case, these segments will be at the minimum length of 0.6 m, as described in Section 3.2. In that case, the maximum time available to generate the next part of the trajectory $t_{\max}$ can be calculated as

$$t_{\max} = \frac{0.6n}{S} - t_{\text{net}}, \tag{5}$$

where S is the speed of the car in meters per second and t_{net} is the network latency in seconds. For example, a maximum speed of 12 km/h, a maximum network latency of 300 ms, and $n = 4$ gives $t_{\max} = 420$ ms to generate the next portion of the trajectory. Notice that the parameter n can be adapted if needed.

4. Results

We have evaluated the performance of our UWB positioning system described in Section 2.2 using our test setup by placing our model car together with two UWB anchors on a parking lot, as shown in Figure 12. We started by manually driving along the edges of the parking lot while drawing a trace on the map to aid in placing the enclosing polygon. There were no cutouts in this test. Then we tried to generate 100 km of trajectory within the area, which succeeded without issues. Next, we used our model car in HIL simulation mode, as described in Section 2.1, running a few experiments to further ensure that the tests are safe to run.

Next, we tried the setup without FI running 10 experiments to ensure that most areas are reached and that the system works nominally without faults present. Figure 13 shows the traces from this experiment. The difference between the UWB position and the RTK-SN position was below 0.6 m for the entire experiment. It is also clear that all traces for the experiments are overlapping at the beginning of the start route (route 0), showing that the return function described in Section 3.3 effectively restores the system state.

Figure 12: Our model car driving on a parking lot with two UWB anchors mounted on tripods in waterproof boxes.

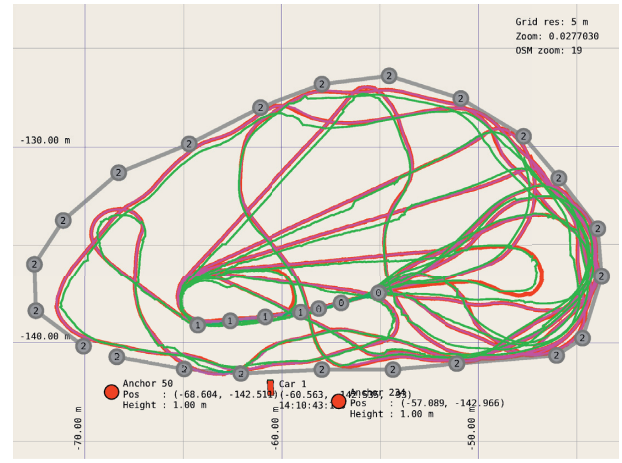

Figure 13: Traces after 10 test campaigns without FI. The RTK-SN traces are shown in magenta and the UWB traces are shown in green.

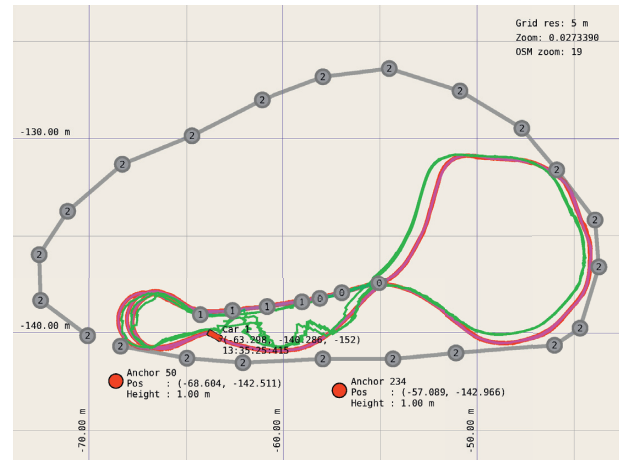

Figure 14: One test repeated three times with FI enabled. The traces for them are overlayed and mostly overlapping. Maximum UWB deviation: 1.7 m.

Then we ran an experiment with FI enabled in the same area, the results are shown in Figure 14. The figure shows overlapping traces for rerunning the same experiment three times using the return functionality described in Section 3.3.

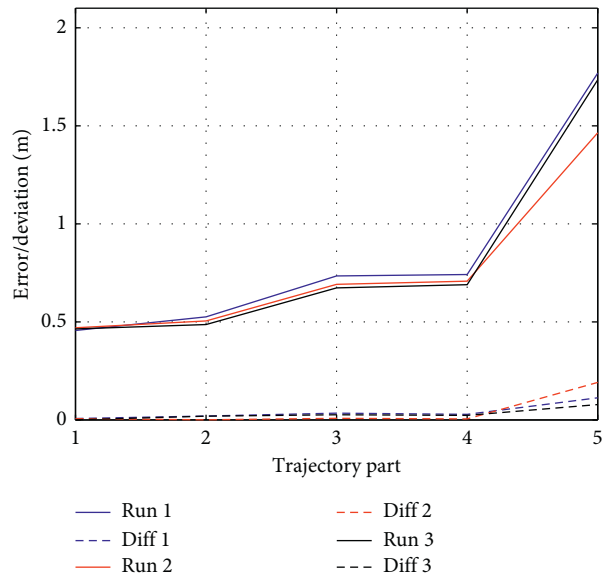

Figure 15: The difference between the UWB and RTK-SN positions for the three reruns of the experiment shown in Figure 14. The dotted lines show the difference from each individual test to the average, indicating the repeatability.

```
(1)  RunSegment(...)
(2)  RunSegment(...)
(3)  AddFault(uwb_travel_dist,AMPLIFICATION,2.5,46,6)//Wheel slip
(4)  RunSegment(...)
(5)  AddFault(uwb_range_234,AMPLIFICATION,2.0,53,4)//UWB Reflection
(6)  RunSegment(...)
(7)  AddFault(uwb_travel_dist,AMPLIFICATION,2.2,44,6)//Wheel slip
(8)  AddFault(uwb_yaw,OFFSET,−18.0,0,2)//Yaw error
(9)  AddFault(uwb_range_50,AMPLIFICATION,4.0,14,8)//UWB Reflection
(10) AddFault(uwb_range_234,AMPLIFICATION,2.0,21,6)//UWB Reflection
(11) RunSegment(...)
```

Algorithm 5

Figure 15 shows the maximum difference between the UWB and the RTK-SN positions for each RunSegment command for the three reruns of the same experiment, as well as the difference between the individual experiments. The experiment consisted of the following commands (Algorithm 5).

Note that each *AddFault* command has a comment after it that describes what the fault represents. Also note that each *AddFault* command has a duration that is short enough to only affect the first *RunSegment* command after it (e.g., the *wheel slip* fault on line 3 only affects the *RunSegment* command on line 4).

As can be seen, the first wheel slip fault brought the deviation up to 0.7 m, and the first reflection fault had the same impact. Together the latter faults injected brought the deviation in position estimates up to around 1.5 m, which made the postcondition of the RunSegment command fail resulting in a failed test case. We got consistent results with less than 0.1 m difference between the runs (except one outlier of 0.2 m towards the end of one run), indicating that this particular combination of faults has a repeatable impact, and that we have good repeatability in our experiments. The fault handling mechanism to handle reflections, as mentioned in Section 2.2, is to truncate the maximum distance an UWB anchor correction is allowed to make to 0.2 m. We were able to repeat the same experiment with different values of this truncation parameter and saw that larger values were better at compensating for wheel slip but were affected more by reflections and smaller values had the opposite effect. Note that significantly more complex fault handling mechanisms can be implemented and evaluated using the same experiment setup, but that is outside the scope of this paper.

We then reran the same experiment without FI enabled three times, as shown in Figures 16 and 17. This time the UWB deviation was below 0.55 m for the entire experiment, and the results were repeatable with a difference of less than 0.1 m between experiments. By observing the green traces from the UWB position, it can be noted that they are almost completely overlapping, with the same kind of deviation consistently when repeating the experiments. This indicates

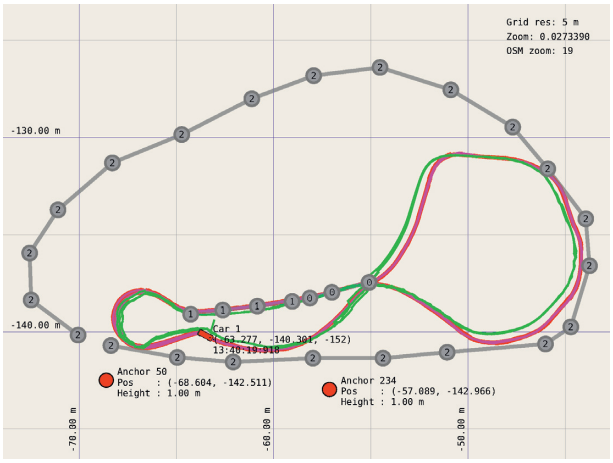

FIGURE 16: The same test as in Figure 14, but with FI disabled. The traces for repetition runs are overlayed, and mostly overlapping, maximum UWB deviation: 0.55 m.

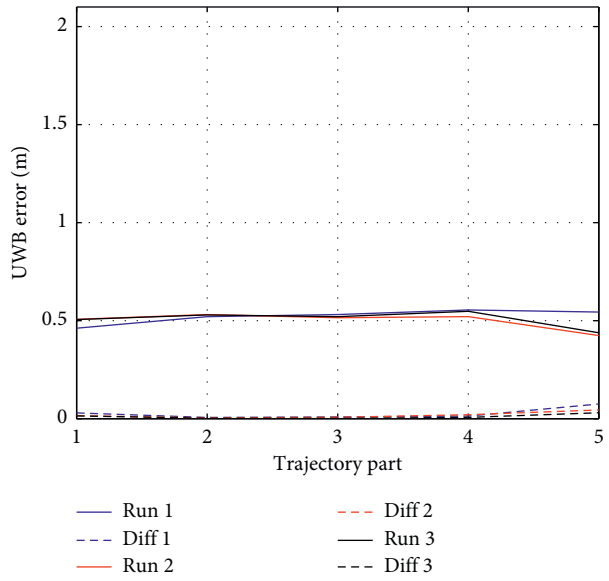

FIGURE 17: The difference between the UWB and RTK-SN positions for three reruns of the experiment shown in Figure 16. The dotted lines show the difference from each individual test to the average, indicating the repeatability.

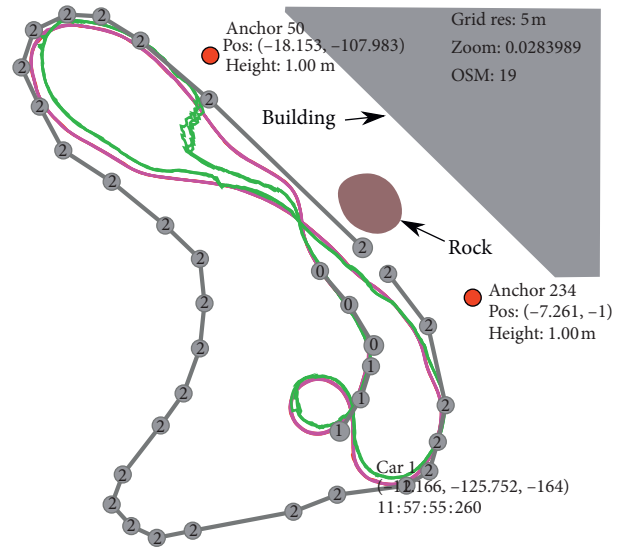

FIGURE 18: Experiment on parking lot with rock without FI, repeated three times. The traces for the repetitions are overlayed, and mostly overlapping, maximum UWB deviation: 2 m.

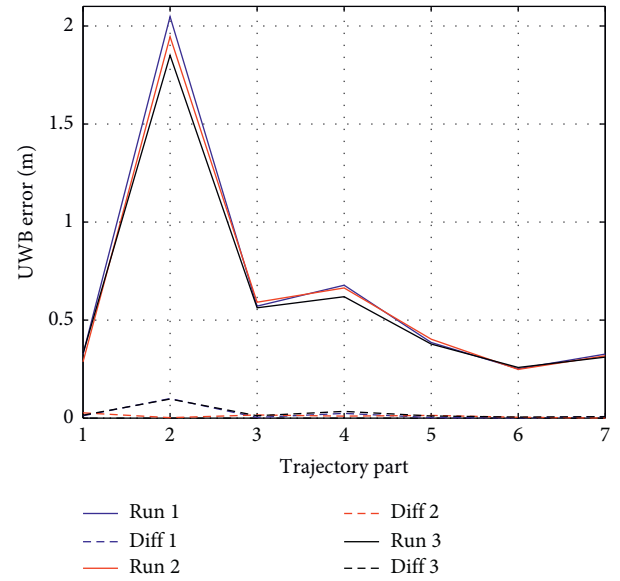

FIGURE 19: The difference between the UWB and RTK-SN positions for the three reruns of the experiment shown in Figure 18. The dotted lines show the difference from each individual test to the average.

that the deviation is of systematic nature such as errors in the anchor positions, incorrect geometry assumptions as described in Section 2.2, and direction-dependent offsets possibly due to UWB antenna gain directionality, as described in our previous work [18].

Next, we set up another experiment on a different parking lot, with a rock blocking the line of sight to one UWB anchor for a portion of the trajectory. There we repeated an experiment that failed without FI three times, with the results shown in Figures 18 and 19. As can be seen, the rock blocks anchor 234 for a section of the trajectory, where the UWB position obtains an offset away from the rock due to the longer measured distance caused by the reflection on the building wall. The three reruns of the experiment had consistent results, with less than 0.15 m difference between reruns, even when the total deviation was around 2 m. The ability to repeat the same experiment again with nearly identical results was a great aid in determining that the deviation was caused by a wall reflection due to the rock blocking the line of sight.

Methods to deal with this type of problems are to alter the anchor placement, add more anchors, and/or improving the fault handling mechanisms of the software.

5. Conclusion

We have presented a novel approach for automatically testing real-time hardware with techniques from PBT and FI. The real-time nature of the task brought challenges such as timing, safety, and repeatability. In the process of addressing these challenges, we have developed a novel method of generating safe and drivable random trajectories respecting the geometry of the car as well as the available area on the map. We also took advantage of this trajectory generator together with a custom trajectory shortening method to generate a drivable trajectory from an arbitrary position to a defined start position and heading for consistently resetting the state of our SUT. Our random trajectory generator also provides a guarantee for each generated part that it is possible to generate an additional trajectory with a specified length after it, which is essential for the safety during tests. Further, we incorporated fault injection in our test setup to make it suitable for testing functional as well as nonfunctional requirements.

With this test setup in place, we developed and tested a low-cost UWB-based positioning system for our self-driving model car. We tested this system against the existing RTK-SN positioning system on the model car and found several interesting randomly generated test cases with and without injected faults that led to failures (deviations exceeding 1 m between the positions based on UWB and RTK-SN). By replaying the test cases many times and comparing the results, we were able to identify causes for the failures and suggest improvements to handle them.

To the best of our knowledge, this is a novel approach for automatically generating tests for complex real-time systems, with regard to safety, timing, and repeatability of conducted experiments. Our work can be used as a basis for testing a variety of real-time systems extensively on, e.g., test tracks dealing with road vehicles, or when testing mobile robots in various situations.

Data Availability

The raw data used to generate the plots and additional data related to the experiments presented in this paper can be found at https://github.com/vedderb/rise_sdvp/tree/master/Misc/Test%20Data/Automated_testing_2019. All software applications used are available as open source from github https://github.com/vedderb/rise_sdvp as well as free and open hardware designs of the UWB anchors.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research has been funded through EISIGS (grants from the Knowledge Foundation), by Vinnova via the FFI project Chronos step 2 and through the PRoPART EU project (Grant agreement no: 776307).

References

[1] B. Vedder, J. Vinter, and M. Jonsson, "Using simulation, fault injection and property-based testing to evaluate collision avoidance of a quadcopter system," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 104–111, Rio de Janeiro, Brazil, June 2015.

[2] A. Soltani and F. Assadian, "A hardware-in-the-loop facility for integrated vehicle dynamics control system design and validation," *IFAC-PapersOnLine*, vol. 49, no. 21, pp. 32–38, 2016.

[3] A. Mouzakitis, D. Copp, R. Parker, and K. Burnham, "Hardware-in-the-loop system for testing automotive ecu diagnostic software," *Measurement and Control*, vol. 42, no. 8, pp. 238–245, 2009.

[4] B. Vedder, J. Vinter, and M. Jonsson, "A low-cost model vehicle testbed with accurate positioning for autonomous driving," *Journal of Robotics*, vol. 2018, Article ID 4907536, 10 pages, 2018.

[5] R. Nilsson, *ScalaCheck: The Definitive Guide*, Artima Inc., Walnut Creek, CA, USA, 2014.

[6] J. Derrick, N. Walkinshaw, T. Arts et al., "Property-based testing—the ProTest project," in *Formal Methods for Components and Objects. Lecture Notes in Computer Science*, F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds., vol. 6286, pp. 250–271, Springer, Berlin, Germany, 2010.

[7] R. K. Iyer, "Experimental evaluation," in *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing, ser. FTCS'95*, pp. 115–132, IEEE Computer Society, Pasadena, CA, USA, June 1995.

[8] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool," in *Proceedings of the Twenty-Fourth International Symposium on Fault-Tolerant Computing*, pp. 66–75, Austin, TX, USA, June 1994.

[9] J. Vinter, L. Bromander, P. Raistrick, and H. Edler, "FISCADE—a fault injection tool for SCADE models," in *Proceedings of the Institution of Engineering and Technology Conference on Automotive Electronics*, pp. 1–9, Warwick, UK, June 2007.

[10] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pp. 284–293, Munich, Germany, June 1998.

[11] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: generic object-oriented fault injection tool," in *Proceedings of the DSN International Conference on Dependable Systems and Networks*, pp. 83–88, Goteborg, Sweden, July 2001.

[12] B. Vedder, T. Arts, J. Vinter, and M. Jonsson, "Combining fault-injection with property-based testing," in *Proceedings of the International Workshop on Engineering Simulations for Cyber-Physical Systems, ser. ES4CPS'14*, pp. 1–8, ACM, Dresden, Germany, March 2014.

[13] J. Cordes, J. Mössinger, W. Grote, and A. Lapp, "Autosar standardised application interfaces," *ATZautotechnology*, vol. 9, no. 2, pp. 42–45, 2009.

[14] M. Skoglund, T. Petig, B. Vedder, H. Eriksson, and E. M. Schiller, "Static and dynamic performance evaluation of low-cost RTK GPS receivers," in *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*, pp. 16–19, Gothenburg, Sweden, June 2016.

[15] M. Haklay and P. Weber, "OpenStreetMap: user-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, 2008.

[16] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro, "Pure pursuit revisited: field testing of autonomous vehicles in urban areas," in *Proceedings of the IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 7–12, Nagoya, Japan, October 2016.

[17] Decawave, *DWM1000 IEEE 802.15.4 UWB Transceiver Module, DWM1000 Datasheet*, Decawave, Dublin, Ireland, 2016.

[18] B. Vedder, J. Vinter, and M. Jonsson, "Accurate positioning of bicycles for improved safety," in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–6, Las Vegas, NV, USA, January 2018.

[19] P. Olsson, "Testing and verification of adaptive cruise control and collision warning with brake support by using HIL simulations," in *Proceedings of the SAE Technical Paper Series*, SAE International, Detroit, MI, USA, April 2008.

[20] E. Bagalini and M. Violante, "Development of an automated test system for ECU software validation: an industrial experience," in *Proceedings of the 15th Biennial Baltic Electronics Conference (BEC)*, pp. 103–106, Tallinn, Estonia, October 2016.

[21] J. Yoon and C. D. Crane, "Path planning for unmanned ground vehicle in urban parking area," in *Proceedings of the 11th International Conference on Control, Automation and Systems*, pp. 887–892, Gyeonggi-do, South Korea, October 2011.

[22] M. S. M. Hashim and T.-F. Lu, "Time-critical trajectory planning for a car-like robot in unknown environments," in *Proceedings of the IEEE Business Engineering and Industrial Applications Colloquium (BEIAC)*, pp. 836–841, Langkawi, Malaysia, April 2013.