HALMSTAD
UNIVERSITY

On the Design and Testing of Dependable Autonomous Systems

Benjamin Vedder

D O C T O R A L T H E S I S | Halmstad University Dissertations no. 52

On the Design and Testing of Dependable Autonomous Systems
© Benjamin Vedder
Halmstad University Dissertations no. 52
ISBN 978-91-88749-10-9 (printed)
ISBN 978-91-88749-11-6 (pdf)
Publisher: Halmstad University Press, 2018 | www.hh.se/hup
Printer: Media-Tryck, Lund

Abstract

Designing software-intensive embedded systems for dependable autonomous applications is challenging. In addition to fulfilling complex functional requirements, the system must be safe under all operating conditions, even in the presence of faults. The key to achieving this is by simulating and testing the system enough, including possible faults that can be expected, to be confident that it reaches an acceptable level of performance with preserved safety. However, as the complexity of an autonomous system and its application grows, it becomes exponentially more difficult to perform exhaustive testing and explore the full state space, which makes the task a significant challenge.

Property-Based Testing (PBT) is a software testing technique where tests and input stimuli for a system are automatically generated based on specified properties of the system, and it is normally used for testing software libraries. PBT is not a formal proof that the system fulfills the specified properties, but an effective way to find deviations from them. Safety-critical systems that must be able to deal with hardware faults are often tested using Fault Injection (FI) at several abstraction levels. The purpose of FI is to inject faults into a system in order to exercise and evaluate fault handling mechanisms. In this thesis, we utilize techniques from PBT and FI, for automatically testing functional and safety requirements of autonomous system simultaneously. We have done this on both simulations of hardware, and on real-time hardware for autonomous systems. This has been done in the process of developing a quadcopter system with collision avoidance, as well as when developing a self-driving model car. With this work we explore how tests can be auto-generated with techniques from PBT and FI, and how this approach can be used at several abstraction levels during the development of these systems. We also explore which details and design choices have to be considered while developing our simulators and embedded software, to ease testing with our proposed methods.

Acknowledgements

I would like to express my gratitude to my supervisor Magnus Jonsson and my co-supervisor Jonny Vinter for supporting me in the work with this thesis. I would also like to thank my family and friends for being there for me during the period I worked with this thesis. Further, I would like to thank my colleagues at RISE Research Institutes of Sweden and Halmstad University, the members of the PROWESS, KARYON and PRoPART EU Projects, and the members of the CHRONOS step 1 and 2 Vinnova projects. Finally, I would like to thank all co-authors of the included papers for their contribution and support.

This research has been funded through the PROWESS EU project (Grant agreement no: 317820), the KARYON EU project (Grant agreement no: 288195), by VINNOVA via the FFI projects Chronos step 1 and Chronos step 2, through the PROPART EU project (Grant agreement no: 776307) and through EISIGS (grants from the Knowledge Foundation).

List of Included Papers

The following papers, referred to in the text by their Roman numerals, are included in this thesis. The layout of the papers has been reformatted to comply with the rest of the thesis, but the content has not been altered.

PAPER I: **Combining Fault-Injection with Property-Based Testing**
Benjamin Vedder, Thomas Arts, Jonny Vinter and Magnus Jonsson. In Proc. of Engineering Simulations for Cyber-Physical Systems (ES4CPS), Dresden, Germany (2014).
DOI: 10.1145/2559627.2559629

Contribution: This paper presents a research prototype tool named FaultCheck that enables property-based testing tools to use common fault injection techniques directly on C and C++ source code. Benjamin has proposed and developed the tool FaultCheck, performed the experiments, and written most of the paper.

PAPER II: **Towards Collision Avoidance for Commodity Hardware Quadcopters with Ultrasound Localization**
Benjamin Vedder, Henrik Eriksson, Daniel Skarin, Jonny Vinter and Magnus Jonsson. In Proc. of the International Conference on Unmanned Aircraft Systems (ICUAS), Denver, Colorado, USA, (2015).

Contribution: This paper presents a custom quadcopter hardware platform with a novel approach on indoor localization. Benjamin has proposed the idea for the localization method, developed most of the electronics and software, and written most of the paper.

PAPER III: **Using Simulation, Fault Injection and Property-Based Testing to Evaluate Collision Avoidance of a Quadcopter System**
Benjamin Vedder, Jonny Vinter and Magnus Jonsson. In Proc. of Safety and Security of Intelligent Vehicles (SSIV), Rio de Janeiro, Brazil (2015).

Contribution: This paper presents more details about the simulator for the hardware quadcopters presented in Paper II. Benjamin has derived fault models from the hardware, applied the testing platform presented in Paper I on the simulator and written the whole paper with comments from his supervisors.

PAPER IV: **Accurate Positioning of Bicycles for Improved Safety**
Benjamin Vedder, Jonny Vinter and Magnus Jonsson. In Proc. of IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, USA (2018).

Contribution: This paper presents an accurate positioning system for bicycles to improve their safety. Benjamin has written the whole paper with comments from his supervisors. He has also developed and tested the positioning system and compared it to other solutions.

PAPER V: **A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving**
Benjamin Vedder, Jonny Vinter and Magnus Jonsson. Accepted for Publication in Hindawi Journal of Robotics (2018).

Contribution: This paper presents our self-driving model car platform, compares it to other platforms and evaluates its performance. Benjamin has written the whole paper with comments from his supervisors. He has also developed the model car, its algorithms and compared it to other solutions.

PAPER VI: **Automated Testing of Ultra-Wideband Positioning for Autonomous Driving**
Benjamin Vedder, Joel Svensson, Jonny Vinter and Magnus Jonsson. To be submitted (2018)

Contribution: This paper shows how our self-driving model car can be used to test our ultra-wideband positioning system by extending techniques introduced in our previous work. Benjamin has written the majority of the paper with comments from his supervisors. He has also developed the majority of the test setup, performed the experiments and analyzed the results. Joel helped writing about the test setup and performing the experiments.

List of Other Publications

The following papers are related but not included in this thesis.

PAPER VII:	**Composable Safety-Critical Systems Based on Pre-Certified Software Components** Andreas Söderberg and Benjamin Vedder, Proc. of ISSREW (2012).
PAPER VIII:	**A Fault-Injection Prototype for Safety Assessment of V2X Communication** Daniel Skarin, Benjamin Vedder, Rolf Johansson and Henrik Eriksson, Proc. of DEPEND (2014).
PAPER IX:	**SafetyADD: A Tool for Safety-Contract Based Design** Fredrik Warg, Benjamin Vedder, Martin Skoglund and Andreas Söderberg, Proc. of WoSoCer (2014).
PAPER X:	**Static and dynamic performance evaluation of low-cost RTK GPS receivers** Martin Skoglund, Thomas Petig, Benjamin Vedder, Henrik Eriksson and Elad M. Schiller, Proc. IEEE Intelligent Vehicles Symposium (IV) (2016).

Contents

Glossary

ADC	Analog to Digital Converter. 49
AHRS	Attitude and Heading Reference System. 50, 70
API	Application Programming Interface. 27, 136, 137
AR	Augumented Reality. 87, 89
ASIL	Automotive Safety Integrity Level. 34
CAN	Controller Area Network. A communication bus standard for indistrial and automotive applications. 89, 90, 94, 106, 109, 110, 131, 133, 134
Dead reckoning	The process of estimating the current position by advancing a previously determined position using inertial or speed measurements. 110, 112–115
E2E	End-to-End. 2, 12–15, 26, 30, 31, 34, 35, 38, 39, 130
FFT	Fast Fourier Transform. 49
FI	Fault Injection. i, 1–4, 7, 8, 11, 16, 25–27, 29, 32, 39, 67, 69, 77, 80, 129–131, 139–141, 147–153
FOC	Field Oriented Control. See [1]. 110
GNSS	Global Navigation Satellite System. xiv, 9, 45–47, 58, 88, 90, 100, 106, 107, 109–111, 113, 116, 133
GUI	Graphical User Interface. 17, 55, 75–78, 107, 110
HIL	Hardware-In-the-Loop. xi, 3, 11, 18, 106, 110, 129, 130, 133, 140, 147
IMU	Inertial Measurement Unit. A chip that can measure acceleration and angular velocity with three axes each. 10, 16, 17, 46–48, 58, 89, 93, 97, 109, 113, 131, 133–135, 140

ITS	Intelligent Transportation System. 55, 56, 74, 75
LDM	Local Dynamic Map. 53, 54, 72, 74
MAV	Micro Air Vehicle. 45, 47
MPC	Model Predictive Control. See [2]. 105
Odometry	Tracking the rotation of the wheels. In our case this is done by tracking the rotation of the motor. 10, 113, 122
PBT	Property-Based Testing. i, 1–6, 8, 11–13, 16, 18, 25–29, 32, 39, 67, 129, 130, 140, 153
PCB	Printed Circuit Board. 17, 88–91, 94, 95, 105–107, 109, 110, 112, 120, 123, 133
PID	Proportional-Integral-Derivative. 50, 52, 70
PMSM	Permanent Magnet Synchronous Motor. A three-phase motor without brushes, often referred to as BLDC motor. 107, 109, 110
PPM	Pulse-Position Modulation. 48
PPS	Pulse Per Second. A pulse that is emitted with accurate timing in the beginning of each second, synchronized to Global Navigation Satellite System (GNSS) time. 113
RFID	Radio-frequency identification. 88
RTK-SN	Real-Time Kinematic Satellite Navigation. See [3, 4]. ix, 9–11, 16–18, 87–90, 92–98, 105, 107–109, 111–115, 122, 129–131, 133, 139, 148, 149, 151, 153, 154
SLAM	Simultaneous Localization and Mapping. 47
SUT	System Under Test. 2, 10, 27, 67, 69, 74, 77, 79, 82, 129–131, 136, 137, 139, 153
SVM	Space Vector Modulation. See [5]. 110
SWD	Serial Wire Debug. A programming and debugging interface. 109
TMR	Triple Modular Redundancy. 7
ToF	Time of Flight. 10, 47, 49, 94
TWR	Two-Way Ranging. See [6]. 10, 94
UWB	Ultra-Wide Band. ix, 3, 9–11, 16–18, 87, 89, 91, 94–96, 100, 129–136, 139, 141, 147–154

1. Introduction

During the development of software-intensive systems, the aim is to ensure that the software specification is fulfilled. Formally proving that software fulfills its requirements under all conditions can be time consuming and difficult, which is why that is not done for the majority of software. A common way to evaluate software is to test specific corner cases to test whether certain requirements are not fulfilled. When these tests pass it is not a proof that the software is correct, but when a test fails it proves that not all requirements under test are not fulfilled.

While the focus of software testing is to make sure that functional requirements are fulfilled, safety-critical systems also have to deal with non-functional requirements such as fault tolerance. For example, an airbag system has strong requirements that it should not inflate the airbag when there is no collision. For such systems, it is important to ensure that they are safe even when certain faults are present in the system. This can be done with fault handling mechanisms implemented in hardware and/or in software. The purpose of fault handling mechanisms is to detect when something is wrong and, for example, compensate for that fault or bring the system to a safe state.

Dependable mobile systems, such as autonomous cars, are challenging to develop. It is desirable to keep the safety-critical parts as simple as possible to make the required testing manageable, but with more sophisticated active safety functions and autonomous driving the complexity of the safety-critical parts grows significantly. We aim at addressing the challenge of managing this complexity by using techniques commonly found in Property-Based Testing (PBT) [7] and Fault Injection (FI) [8], combining them, and applying them across several abstraction levels during the development of complex mobile systems. To evaluate our approach, we have developed an autonomous quadcopter system with indoor positioning together with a simulator for it, as well as a self-driving model car platform with accurate outdoor positioning. During the development, we investigate how automatic test case generation with techniques from PBT and FI can be applied, both for evaluating functional and non-functional requirements.

1.1 Research Questions

As described above, we focus on developing and testing of complex dependable autonomous systems. We want to automate testing and verification as much as possible, from simulation to testing on real-time hardware. For doing so, we utilize techniques from PBT and FI. This gives rise to the following questions, which we try to answer in this thesis:

Q1 How can knowledge from FI and PBT be combined in a practical automated testing platform?

Q2 How does this testing platform scale when the System Under Test (SUT) becomes more complex?

Q3 What is required, both from the testing platform and from the SUT, to use the same approach with software components and simulations, as well as with tests on real-time hardware?

Q4 How can realistic fault models be derived for a complex, simulated or real-time, SUT, and how can they be injected while performing automated testing?

Q5 What are the challenges when automatically testing moving real-time hardware, and how to deal with them?

1.2 Research Approach

We addressed Q1 by first investigating how PBT and FI can be carried out simultaneously. For doing so, we have developed a tool named FaultCheck, that can be used from PBT tools to utilize common FI techniques while performing PBT. We demonstrate this approach by evaluating a simple End-to-End (E2E) library. To address Q2, Q3 and Q4, we focused on our quadcopter simulator. We have developed the simulator together with the hardware quadcopter system, in order to make it an accurate representation of the actual hardware. For example, our custom positioning system based on ultrasound is represented in the simulator, which enables us to inject realistic position faults into the simulation. We then evaluate the simulated quadcopters by automatically generating tests using a PBT-tool named QuickCheck, and our fault injection tool FaultCheck. With this, we address the challenge of automatically generating test cases that evaluate both functional and safety requirements simultaneously on a relatively complex autonomous system, during simulation.

After the quadcopter simulator, to further address Q3 as well as Q5, we focused on developing a self-driving model vehicle platform. We test our model cars with the same techniques we used for the quadcopter simulator, with the difference that we now are generating test cases for real-time hardware that is moving around during the auto-generated tests, rather than for a simulator that runs purely in software using simulated time. In addition to the challenges with the quadcopter simulator, this brings the new challenges of executing the tests in real-time, restoring the state between tests and keeping the tests safe now that they involve moving hardware.

With this work, we show an approach on developing and testing complex dependable systems, together with simulation and automatic test case generation, all the way from realistic simulations to automatic testing on real-time hardware. During this journey, we encountered challenges such as how to inject realistic faults in simulations and how to deal with automatically testing moving real-time hardware, and propose approaches on addressing these challenges.

1.3 Research Contributions

The main contributions of this thesis, attempting to answer the research questions, are:

C1 We present methods on how FI and PBT can be combined in order to test fault tolerance and functional requirements simultaneously. The ideas are implemented in a tool named FaultCheck, which enables PBT-tools such as QuickCheck to inject faults based on common fault models into C and C++ code (Paper I). This contribution addresses **Q1**.

C2 We develop a quadcopter system with a novel approach for indoor positioning, together with a closely-connected simulator. The simulator represents many details about the hardware, the positioning system and the communication between the quadcopters, which enables us to carry out realistic fault injection with our FaultCheck tool. We show several design choices from the development that makes the simulator convenient to use with our automated testing setup (Paper II and Paper III). This contribution addresses **Q2** and **Q3**.

C3 We show how to derive realistic fault models, both for our quadcopter simulator (Paper III) and for our Ultra-Wide Band (UWB) positioning system (Paper VI), and how to inject them using our testing platform. This gives us confidence that fault models can be derived in a similar way for a wide range of complex systems from different domains. This contribution addresses **Q4**.

C4 We show how to develop a complex model car that is suitable to be used with our testing approach. By making it possible to activate different parts of the positioning system while simulating others on the hardware, it is possible to first develop tests with Hardware-In-the-Loop (HIL), and then run the same automated testing setup in a real scenario with minimal changes (Paper IV and Paper V). This contribution addresses **Q3**.

C5 We show how to test the components of the redundant positioning systems of the model car against each other with auto-generated test cases (Paper VI). Compared to the tests conducted in Paper I to Paper III, the automatic tests are running in real time with the extra challenges that come from that. Namely, tests have to be constructed such that no hardware gets destroyed when they fail, timing is critical and fewer tests can be run as they are executed in real time as opposed to simulated time. We also address the challenge of restoring the state of the full-scale system between tests, by automatically driving the model car back to the start position while avoiding obstacles (Paper VI). This contribution addresses **Q2**, **Q4** and **Q5**.

1.4 Thesis Structure

The rest of this thesis is organized as follows: In the next chapter we introduce PBT and FI, and we present an overview about the accurate positioning technologies that are essential for the work on our self-driving model vehicle platform. The chapter after that summarizes the included papers, and explains how they are connected. It also shows additional results related to Paper I, that were not included in the paper. Finally, the papers with details about the conducted research are appended.

2. Background

2.1 Property-Based Testing

Writing software tests is often done in the form of unit tests, where each test case with fixed input data is written manually. In order to create a wide variety of test cases, a technique named PBT can be used [7]. When doing PBT, many "unit tests" with different parameters are automatically generated from the specification of a property. An early lightweight tool that can be used for PBT is the Haskell QuickCheck library [9]. For example, a property that defines that the reverse of a reversed list should be equal to the initial list can be expressed like the following:

```
prop_RevRev xs =
  reverse (reverse xs) == xs
```

where QuickCheck will generate lists *xs* of random sizes and check if that property returns true for all of them.

Because of its popularity, the QuickCheck algorithm has been ported to many programming languages, including Scala [10], Erlang [11], Python [12] and Java [13]. ScalaCheck, which is the QuickCheck algorithm implemented in the Scala programming language, is integrated in the Scala testing framework ScalaTest [14] and used by prominent Scala projects such as the Akka concurrency framework [15]. One commercially available implementation of QuickCheck is Erlang QuickCheck [11], which has been used for the testing of large scale systems [16, 17]. While working on the first three papers included in this thesis we had close collaboration with QuviQ[1], the company behind Erlang QuickCheck, which is why we used Erlang QuickCheck in them. In the last paper we evaluated ScalaCheck, which has the advantage of being open source, while providing all the functionality we need. ScalaCheck allowed us to study how it works internally, and to conveniently take advantage of the high performance of the Java virtual machine for implementing resource-intensive trajectory generation algorithms.

2.1.1 Testing Stateful Software

Many pieces of software behave according to their internal state. For example a counter, with the functions *Increment*, *Decrement* and *Get*, will return a count when running *Get* that depends on the initial count and how many *Increment*

[1]http://quviq.com/

and *Decrement* commands have been run. To test systems like that, several PBT tools, such as Erlang QuickCheck and ScalaCheck, have implemented a testing method that is aware of the system state. With this testing method, a sequence of commands is generated where each command has a generator, a precondition, a postcondition and a way to update the system state. The purpose of them are as follows:

Generator: The generator is responsible for generating input data for the command. It has access to the state of the system and can adjust the generation of input data according to the state, if necessary.

Precondition: The precondition must be true in order to allow this command to be run. For the example with the counter, it could be the case that a *Decrement* command only is allowed when the counter is non-zero. This can be expressed in the precondition which can throw away this command if the count value of the system state is zero.

Postcondition: The postcondition has to be true in order for the test to pass. The postcondition check also has access to the system state and can decide whether the behaviour is correct depending on the state. For the example with the counter, the postcondition for the *Get* command is that the result should be equal to the count state of the system.

State update: After each command, the state of the system can be updated if necessary. For example, the *Increment* command for the counter should add one to the count value of the system state.

Because the systems we work with in this thesis are inherently stateful, stateful PBT is what we are using.

2.1.2 Shrinking

One feature most PBT tools have is the ability to do shrinking. Shrinking means that when a failing test case is found, the PBT tool tries to reduce it to a smaller failing test case. For non-stateful testing, an example is that a property that takes a list as an input argument fails when the list has duplicate elements. Since the lists are randomly generated, the first list with duplicates might have many other elements. In that case shrinking tries to remove one element from the list at a time until the smallest list that causes the same failure is found, which should be a list with just two elements with the same value.

When it comes to testing of stateful systems, shrinking will first remove one command at a time from the generated command sequence that led to a failure and then try to make the arguments (if any) smaller. For example, if a counter with an initial count of 0 fails when the command *Decrement* is run when the count is zero, a command sequence of [*Increment*, *Decrement*, *Decrement*] will be reduced to just the command *Decrement* since that is enough to trigger the failure. Shrinking can be complicated for stateful systems since removing commands affects the system state and might make the whole command sequence invalid. This can happen if the removal of one command affects the system state in such a way that the next command would not have been generated because its precondition is not true after the removal. Therefore, it is common that the shrinking process has to be adjusted manually to make it work with stateful systems.

2.2 Fault Injection

Collecting statistical data about the effects of faults under normal operation of a system is often not feasible during development, because faults can appear with very low frequencies. FI is a method where the occurrence of faults is accelerated in order to exercise and evaluate fault handling mechanisms [8]. FI is highly recommended in the safety standard IEC 61508 [18] when the required diagnostics coverage is at least 90%. The automotive functional safety standard ISO 26262 [19] recommends FI for ASIL A & B applications and highly recommends FI for ASIL C & D applications.

Traditional targets for FI have been hardware such as microprocessors and memories. One example of physical FI is scan-chain implemented FI [20] where the internal state of a microprocessor can be observed and controlled in a detailed way. The purpose is to evaluate what happens when something, such as ionizing particles, affects the internal state of the microprocessor. In other literature, heavy-ion radiation has been used directly on microprocessors to evaluate how the execution is affected by transient errors that affect the internal state of different registers [21]. Radiation is present everywhere, but in some environments such as airplanes and in space the levels are significantly higher than on ground and therefore these types of faults become even more evident. In order to deal with faults that affect the internal state of microprocessors, it may be necessary to build special architectures that are designed to detect and handle such faults. For example, the fault tolerant version of the Leon microprocessor [22] has several internal fault detection mechanisms, such as Triple Modular Redundancy (TMR) on particular hardware blocks.

Another type of physical FI is pin-level FI [23] where external faults are emulated by affecting the pins of, for example, microprocessors. Dealing with pin-level faults does not require special microprocessor architectures, but other fault handling mechanisms have to be used.

Hardware faults, emulated by physical FI as presented above, can also be emulated by injecting faults into models of hardware [24, 25] or directly into software [26, 27]. Software-implemented FI introduces some overhead to the execution, but has the advantage that it can be done early in the development process before the final hardware is available. Other advantages are that setting up and carrying out numerous experiments can be easier.

2.2.1 Golden Run

A *golden run* is a recorded reference run used during fault injection for which no faults are injected. For example, when testing a speed controller for a vehicle, the test can first be run without faults using a selected input vector while the output of the speed controller as well as the state of the vehicle is recorded. After that, the test can be run again while faults are injected and the speed controller output and the vehicle state are compared to the golden run. This way faults, or combinations of faults, that cause unacceptable deviation from the golden run for the input vector of this test can be identified.

For a given system, there may be many invalid input vectors and a deviation from the golden run can be defined in different ways. Therefore, creating the golden run can be a manual process and the possibility to test with a wide variety of different golden runs can be limited. This is one of the limitations we would like to address by using PBT together with FI. By creating random generators and postconditions (see Section 2.1.1) that are aware of the system state and have a model of the system, we would like to automatically generate golden runs and be able to check whether the tests with injected faults pass or fail for each of those automatically generated golden runs. This way, we will be able to do FI with a wide variety of golden runs generated automatically.

2.2.2 Characteristics of Fault Injection

There are several properties that can be defined to describe different characteristics of FI. *Reachability* describes how many possible locations of faults can be reached in a system. For example, a simulation-based FI tool for VHDL models of microprocessors has higher reachability than a scan-chain implemented FI tool that only reaches registers and memory locations. *Repeatability* describes how well the same experiment can be repeated. Heavy-ion FI is an example where it is difficult to repeat the same experiment, while software-implemented FI makes repeating the same experiment easier. *Controllability* describes how well the location of faults in space and time can be controlled. Even here, heavy-ion FI is an example of low controllability. Further, *intrusiveness* describes how much undesired impact the FI has on the system. Software-implemented FI can introduce overhead in execution speed, while heavy-ion FI does not have any intrusiveness at all. *Observability* refers to how well the effect of faults on the system can be measured. Finally, *effectiveness* describes how well the FI technique is able to trigger fault handling mechanisms and *efficiency* defines the amount of effort required to conduct and repeat the FI experiments.

In this thesis, where we combine PBT with FI, we aim to have good reachability, repeatability, controllability, observability, efficiency and effectiveness. However, our technique will have some intrusiveness on the system that we are testing. In the later part of the thesis where we perform automatic testing of real-time hardware, good repeatability is more challenging to achieve as random variations in the environment affect the experiments. Bringing the system back to a known state also adds to this difficulty, as it involves automatically driving our model cars back to an initial position from the random position the previous experiment made it end up in. Our results regarding repeatability while automatically testing real-time hardware are promising, and we demonstrate techniques to deal with the challenges.

2.3 Accurate Positioning

For the development of the self-driving model car in Paper V to Paper VI, as well as for the experiments in Paper IV, it is essential to have knowledge about the positioning techniques used. Therefore, we include a brief introduction to them here. Our positioning techniques are based on two technologies: Real-Time Kinematic Satellite Navigation (RTK-SN) and UWB ranging.

2.3.1 RTK-SN

Global Navigation Satellite Systems (GNSSs) is a widely used technology today, and we rely on it daily. It works by passively receiving code words modulated onto a carrier signal from at least four satellites in space, and determining the distance to them by comparing the time when the code words were sent to the time they were received. Based on that distance, and the known satellite orbits, the position of the receiver can be calculated. The accuracy of GNSS can be improved by using more satellites and compensating for signal delays caused by the atmosphere of the earth, but it will still not be better than within several meters [28]. This is impressive considering that we are measuring how long it takes for light to travel to us from fast moving satellites in space using inexpensive integrated circuits, but it is still not good enough for the accuracy required for autonomous driving and thus for our experiments.

RTK-SN uses the same satellites as conventional GNSS, but in addition to measuring the code modulated onto the carrier, the carrier itself is tracked. Tracking the carrier does not provide a direct measurement of the distance to the satellites, but it provides accurate high-resolution tracking of the movement relative to the satellites. Using this information, together with the same measurements from a base station with a known position within 10 km, position accuracies of around 1 cm can be achieved [28].

Traditional RTK-SN systems have been rather expensive, but recently more cost effective solutions have shown up. RTKLIB [3] is a library that allows using rather inexpensive receivers that output raw code and carrier-phase measurements, such as the Ublox M8T[1], both as a base station and as a rover to be positioned. With these measurements, RTKLIB provides a processed position solution with around 1 cm accuracy. More recently, affordable receivers that process the data internally, such as the Ublox M8P[2] have also become available. The accuracy of the lower-cost solution tends to be on par with higher-cost solutions, such as the OXTS RT1003[3], but their convergence time and position output rate is lower. For example, the RT1003 has an output rate

[1]https://www.u-blox.com/en/product/neolea-m8t-series

[2]https://www.u-blox.com/en/product/neo-m8p-series

[3]https://www.oxts.com/products/rt1003/

of 100 Hz, whereas the M8P has an output rate of 5 Hz where each sample is around 100 ms old. In our model vehicle platform, we use odometry and an Inertial Measurement Unit (IMU) to get 100 Hz output rate, and to compensate for the sample age. More details about this can be found in Paper V.

2.3.2 UWB Ranging

In Paper IV we used UWB ranging as a reference to evaluate RTK-SN under poor conditions, and in Paper VI we used it as a SUT to demonstrate our approach on automated test case generation on real-time hardware. UWB ranging works by sending packets between radios using very wide bandwidth. These packets can be accurately timestamped because they are short in time due to the wide bandwidth, and the timestamps can then be used to calculate the distance between the radios. As is the case with RTK-SN, low-cost solutions have only become available recently.

UWB ranging works by sending packets between two transceivers with high-resolution clocks and comparing the timestamps of when the packets were sent and received. This way the Time of Flight (ToF) can be calculated and divided by the speed of light to get the distance between the radios. As the ToF is short compared to the time it takes to transmit and receive the packets, a small difference in the clock speed can cause large errors. These errors can be compensated for using various methods, for example with a scheme called Two-Way Ranging (TWR) [6]. With TWR, three packets with certain timestamps are sent between two radios, where the timestamps can be used to estimate the clock speed difference between the radios. TWR performs well when the clock speeds of the transceivers differ slightly, as long as the clock drift during the ranging operation is small enough. We have been using the Decawave DWM1000 [29] in our experiments, which can achieve accuracies of around 10 cm when using TWR together with signal attenuation compensation [30].

RTK-SN achieves better accuracies compared to UWB ranging, but the error sources are different and independent. For RTK-SN a clear view of a large enough portion of the sky is required, whereas UWB ranging only requires the radios to be close enough to each other have a connection. Therefore, UWB can be used indoors as well. With line-of-sight conditions, the DWM1000 modules we have been using, can reach up tp 200 m. Another important difference between UWB ranging and RTK-SN is the convergence time. RTK-SN can take up to 5 minutes to converge after loosing track of most satellites, whereas UWB does not need to converge at all and can be used from the first sample. This makes these systems suitable to complement each other to cover more areas, or to provide redundant positioning with higher integrity.

3. Summary of Papers

Included in this thesis are six papers that present the research and experiments that have been carried out. In Paper I we introduce FI and PBT, and present a tool named FaultCheck that can be used from PBT tools like QuickCheck in order to utilize common FI techniques. In Paper II we present a novel hardware quadcopter platform and a simulator that we have developed such that it integrates well with our testing platform. After that, in Paper III, we present details about how we have tested our quadcopter simulator using FaultCheck and QuickCheck.

In Paper IV we introduce RTK-SN and UWB-based positioning while studying how RTK-SN can be used to implement active safety on bicycles. These technologies are essential for the work carried out in the next papers. In Paper V we document the implementation and performance of a self-driving model car that we have developed, which is suitable for use with auto-generated tests both with HIL and on full hardware. Finally, in Paper VI we equip the model car with UWB positioning in addition to RTK-SN to get a redundant positioning system. We then verify the performance and fault handling capability of the UWB positioning system against the RTK-SN positioning system by auto-generating test trajectories and following them while injecting various faults. This way we show how automatic test-case generation combined with FI can be used on several abstraction levels throughout the design and development of software-intensive dependable autonomous systems.

3.1 Paper I: Combining Fault-Injection with Property-Based Testing

This paper introduces the areas of FI and PBT, and proposes a tool that enables PBT tools to use fault injection directly on C and C++ source code. The tool is named FaultCheck, and is intended to be used from PBT tools as shown in Figure 1. FaultCheck is a library written in C++ with a wrapper in C, so that it can be linked against from C and C++ programs that are to be tested. It is used by adding probes to variables in the programs to be tested, where any number of faults can be injected. FaultCheck supports fault models such as *bitflip, offset, stuck at* and *amplification* that can be triggered at certain times or permanently. Faults can be activated simultaneously, even on the same probes, with different types of triggers independently of each other.

To control the probes, FaultCheck provides an interface that can be accessed from the PBT tool in the same way as it accesses the rest of the program that

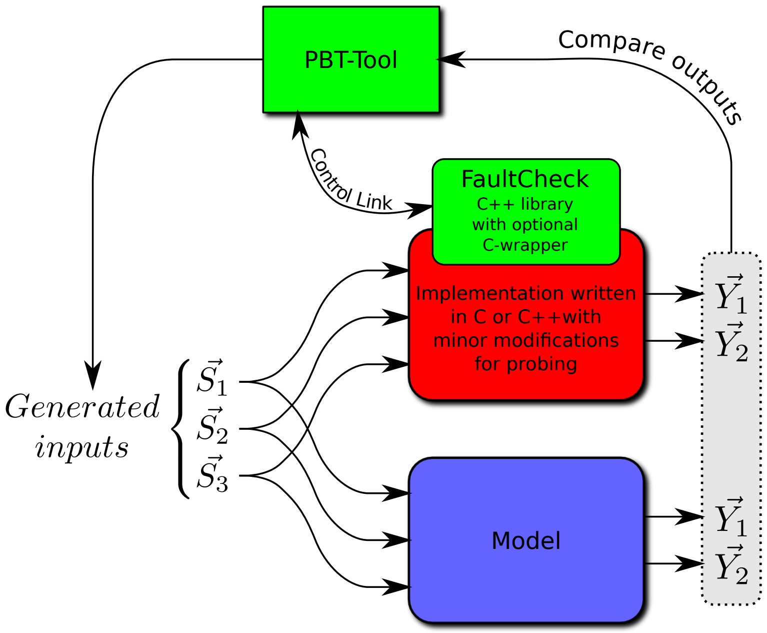


Figure 1: FaultCheck used together with a PBT-tool.

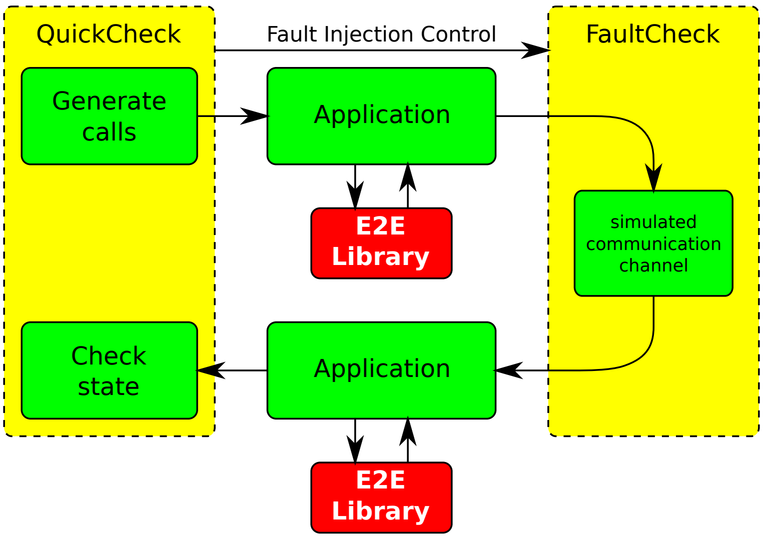
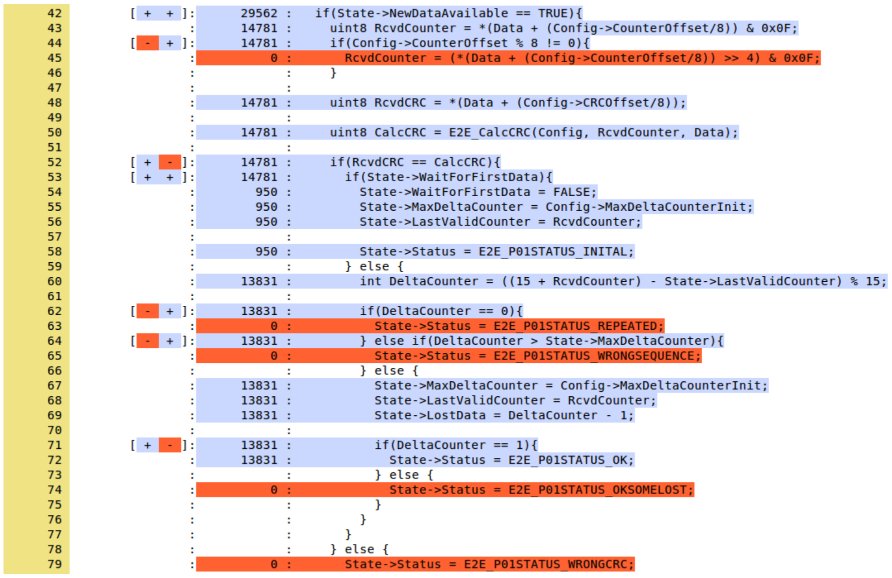


Figure 2: Experiment setup to evaluate the E2E library.



```
[ +  + ]:     29562 :   if(State->NewDataAvailable == TRUE){
        :     14781 :     uint8 RcvdCounter = *(Data + (Config->CounterOffset/8)) & 0x0F;
[ -  + ]:     14781 :     if(Config->CounterOffset % 8 != 0){
        :         0 :       RcvdCounter = (*(Data + (Config->CounterOffset/8)) >> 4) & 0x0F;
        :           :     }
        :           :
        :     14781 :     uint8 RcvdCRC = *(Data + (Config->CRCOffset/8));
        :           :
        :     14781 :     uint8 CalcCRC = E2E_CalcCRC(Config, RcvdCounter, Data);
        :           :
[ +  - ]:     14781 :     if(RcvdCRC == CalcCRC){
[ +  + ]:     14781 :       if(State->WaitForFirstData){
        :       950 :         State->WaitForFirstData = FALSE;
        :       950 :         State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
        :       950 :         State->LastValidCounter = RcvdCounter;
        :           :
        :       950 :         State->Status = E2E_P01STATUS_INITAL;
        :           :       } else {
        :     13831 :         int DeltaCounter = ((15 + RcvdCounter) - State->LastValidCounter) % 15;
        :           :
[ -  + ]:     13831 :         if(DeltaCounter == 0){
        :         0 :           State->Status = E2E_P01STATUS_REPEATED;
[ -  + ]:     13831 :         } else if(DeltaCounter > State->MaxDeltaCounter){
        :         0 :           State->Status = E2E_P01STATUS_WRONGSEQUENCE;
        :           :         } else {
        :     13831 :           State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
        :     13831 :           State->LastValidCounter = RcvdCounter;
        :     13831 :           State->LostData = DeltaCounter - 1;
        :           :
[ +  - ]:     13831 :           if(DeltaCounter == 1){
        :     13831 :             State->Status = E2E_P01STATUS_OK;
        :           :           } else {
        :         0 :             State->Status = E2E_P01STATUS_OKSOMELOST;
        :           :           }
        :           :         }
        :           :       }
        :           :     } else {
        :         0 :       State->Status = E2E_P01STATUS_WRONGCRC;
```

Figure 3: The code coverage of the E2E library when running the experiment without injecting any faults.

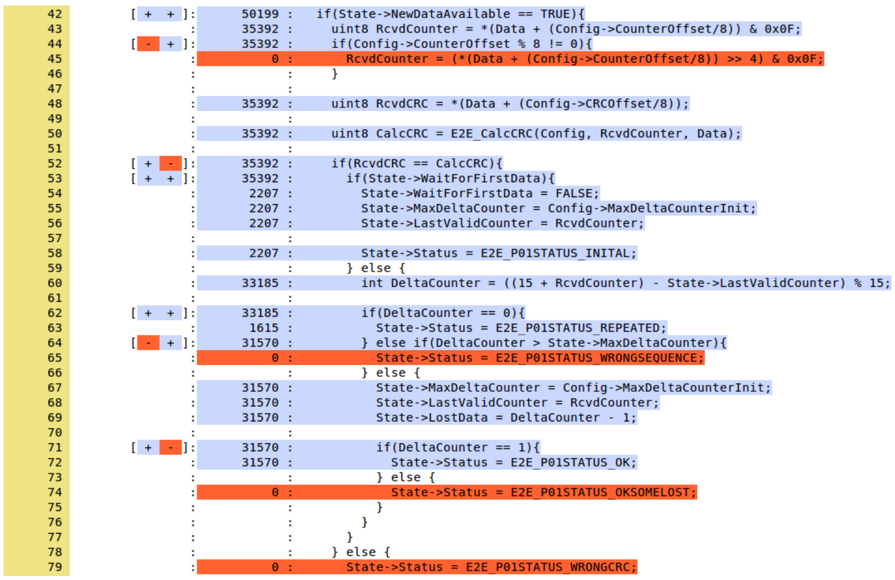
is being tested. The different probes are addressed by using string identifiers, which are stored in a hash table in FaultCheck. For each identifier, any number of faults with different triggering settings can be added.

FaultCheck has e.g. the ability to emulate a communication channel where communication faults can be injected. The communication channel is fed with packets and packets can be fetched from it. To corrupt the packets, all fault models mentioned previously are supported on the bytes of the packets, and other communication faults such as *packet drop* and *packet repetition* are supported. One instance of FaultCheck can simulate any number of communication channels and the fault injection can be controlled from the PBT tool in the same way as the probing faults are controlled.

In this paper, we have used the AUTOSAR E2E library [31] as an example application and the PBT tool Erlang QuickCheck to control FaultCheck, as shown in Figure 2.

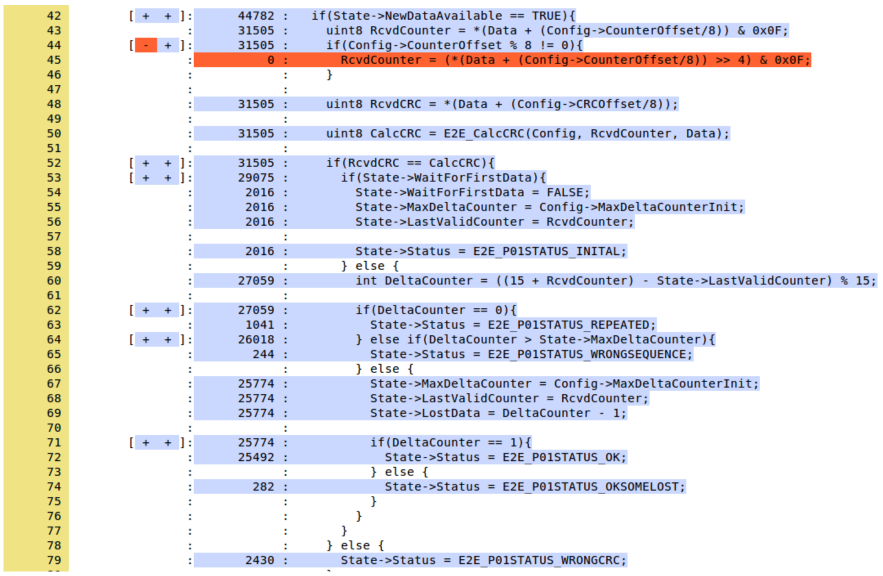
In addition to what we presented in the paper, we also used the gcov and lcov tools[1] to annotate the code of the E2E library with which lines of code have been executed in order to determine which fault handling mechanisms got activated while certain faults were injected. In Figure 3 it can be seen that when the experiment is run without injecting faults, the state is never set to any of the faults that can be detected by the E2E library. Figure 4 shows that when the experiment is run with only repetition faults, the *E2E_P01STATUS_REPEATED* fault code gets activated, but no other fault codes. Finally, Figure 5 shows that when the experiment is run with all fault injection active,

[1]http://ltp.sourceforge.net/coverage/



```
[ +  + ]:   50199 :   if(State->NewDataAvailable == TRUE){
         :   35392 :     uint8 RcvdCounter = *(Data + (Config->CounterOffset/8)) & 0x0F;
[ -  + ]:   35392 :     if(Config->CounterOffset % 8 != 0){
         :       0 :       RcvdCounter = (*(Data + (Config->CounterOffset/8)) >> 4) & 0x0F;
         :         :     }
         :         :
         :   35392 :     uint8 RcvdCRC = *(Data + (Config->CRCOffset/8));
         :         :
         :   35392 :     uint8 CalcCRC = E2E_CalcCRC(Config, RcvdCounter, Data);
         :         :
[ +  - ]:   35392 :     if(RcvdCRC == CalcCRC){
[ +  + ]:   35392 :       if(State->WaitForFirstData){
         :    2207 :         State->WaitForFirstData = FALSE;
         :    2207 :         State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
         :    2207 :         State->LastValidCounter = RcvdCounter;
         :         :
         :    2207 :         State->Status = E2E_P01STATUS_INITAL;
         :         :       } else {
         :   33185 :         int DeltaCounter = ((15 + RcvdCounter) - State->LastValidCounter) % 15;
         :         :
[ +  + ]:   33185 :         if(DeltaCounter == 0){
         :    1615 :           State->Status = E2E_P01STATUS_REPEATED;
[ -  + ]:   31570 :         } else if(DeltaCounter > State->MaxDeltaCounter){
         :       0 :           State->Status = E2E_P01STATUS_WRONGSEQUENCE;
         :         :         } else {
         :   31570 :           State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
         :   31570 :           State->LastValidCounter = RcvdCounter;
         :   31570 :           State->LostData = DeltaCounter - 1;
         :         :
[ +  - ]:   31570 :           if(DeltaCounter == 1){
         :   31570 :             State->Status = E2E_P01STATUS_OK;
         :         :           } else {
         :       0 :             State->Status = E2E_P01STATUS_OKSOMELOST;
         :         :           }
         :         :         }
         :         :       }
         :         :     } else {
         :       0 :       State->Status = E2E_P01STATUS_WRONGCRC;
```

Figure 4: The code coverage of the E2E library when running the experiment when only injecting repetition communication faults.



```
[ +  + ]:   44782 :   if(State->NewDataAvailable == TRUE){
         :   31505 :     uint8 RcvdCounter = *(Data + (Config->CounterOffset/8)) & 0x0F;
[ -  + ]:   31505 :     if(Config->CounterOffset % 8 != 0){
         :       0 :       RcvdCounter = (*(Data + (Config->CounterOffset/8)) >> 4) & 0x0F;
         :         :     }
         :         :
         :   31505 :     uint8 RcvdCRC = *(Data + (Config->CRCOffset/8));
         :         :
         :   31505 :     uint8 CalcCRC = E2E_CalcCRC(Config, RcvdCounter, Data);
         :         :
[ +  + ]:   31505 :     if(RcvdCRC == CalcCRC){
[ +  + ]:   29075 :       if(State->WaitForFirstData){
         :    2016 :         State->WaitForFirstData = FALSE;
         :    2016 :         State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
         :    2016 :         State->LastValidCounter = RcvdCounter;
         :         :
         :    2016 :         State->Status = E2E_P01STATUS_INITAL;
         :         :       } else {
         :   27059 :         int DeltaCounter = ((15 + RcvdCounter) - State->LastValidCounter) % 15;
         :         :
[ +  + ]:   27059 :         if(DeltaCounter == 0){
         :    1041 :           State->Status = E2E_P01STATUS_REPEATED;
[ +  + ]:   26018 :         } else if(DeltaCounter > State->MaxDeltaCounter){
         :     244 :           State->Status = E2E_P01STATUS_WRONGSEQUENCE;
         :         :         } else {
         :   25774 :           State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
         :   25774 :           State->LastValidCounter = RcvdCounter;
         :   25774 :           State->LostData = DeltaCounter - 1;
         :         :
[ +  + ]:   25774 :           if(DeltaCounter == 1){
         :   25492 :             State->Status = E2E_P01STATUS_OK;
         :         :           } else {
         :     282 :             State->Status = E2E_P01STATUS_OKSOMELOST;
         :         :           }
         :         :         }
         :         :       }
         :         :     } else {
         :    2430 :       State->Status = E2E_P01STATUS_WRONGCRC;
```

Figure 5: The code coverage of the E2E library when running the experiment while injecting all possible communication faults.

all fault codes of the E2E library become active several times. This shows that we need to inject several types of faults in order to exercise all fault handling mechanisms of the AUTOSAR E2E library, and this can be done effectively using FaultCheck and QuickCheck.

3.2 Paper II: Towards Collision Avoidance for Commodity Hardware Quadcopters with Ultrasound Localization

This paper presents a quadcopter platform built from inexpensive hardware that is able to do localization based on ultrasound only. One of our goals was to design the quadcopter platform in such a way that it is easy to transport and can be set up in new locations in less than 15 minutes. To our knowledge, our quadcopters are the only ones that can do a stable hover relying only on ultrasound localization in addition to the inertial sensors on the copters. A similar quadcopter platform has been developed by J. Eckert et al. [32, 33] and relies on ultrasound for localization. However, their quadcopters also rely on optical flow sensors directed towards the floor and ceiling, and their ultrasound system has significantly shorter range than ours.

In addition to creating the hardware quadcopter platform we have developed a simulator for the quadcopters that shares much code with the firmware on the hardware quadcopters. The simulator also emulates our localization system and communication between the quadcopters. Using the simulator, we developed a collision-avoidance mechanism based on communication between the quadcopters as well as risk contours. To test the simulator, we used our testing platform described in Paper I. This enabled us to randomly generate thousands of simulations and randomly inject faults while running them in order to evaluate the localization and collision avoidance algorithms.

3.3 Paper III: Using Simulation, Fault Injection and Property-Based Testing to Evaluate Collision Avoidance of a Quadcopter System

This paper describes more details about how our testing platform, based on FaultCheck and QuickCheck, can be used on the quadcopter simulator introduced in Paper II. We show how to derive realistic fault models based on the hardware quadcopter platform and how FaultCheck can be used to represent and inject them in the differential equations that are continuously solved by the quadcopter simulator. Further, we present all tools that we developed around the quadcopter simulator and how they are connected. We also show how the QuickCheck model works that we use to automatically generate tests for the system and how FaultCheck is used from the quadcopter simulator.

In order to visualize failed test cases, we present a way in which we can modify the command sequences generated by QuickCheck so that the test case can be played back smoothly in real time and visualized on a map during the playback. This is very useful since it is difficult to understand what went wrong when just looking at the sequence of commands that led to a failure.

Since the quadcopter simulator is complex and the effort and overhead required to test it with FaultCheck and QuickCheck was relatively small, we have confidence that PBT can be used together with FI to test a wide range of safety-critical systems efficiently.

3.4 Paper IV: Accurate Positioning of Bicycles for Improved Safety

In this paper we have evaluated the use of inexpensive RTK-SN receivers with multiple satellite constellations together with dead reckoning for accurate positioning of bicycles to enable active safety functions such as collision warnings. This is a continuation of previous work (not included in this thesis) were we concluded that RTK-SN alone is not sufficient in moderately dense urban areas since buildings and other obstructions degrade the performance of RTK-SN significantly [4]. We have added odometry to the positioning system as well as extending RTK-SN with multiple satellite constellations to deal with situations where the view of the sky is poor and thus fewer satellites are in view. To verify the performance of the positioning system we have used UWB radios as an independent positioning system to compare against while testing during poor conditions for RTK-SN. We were able to verify that adding dead reckoning and multiple satellite constellations improves the performance significantly under poor conditions and makes the positioning system more useful for active safety systems.

This work helped us understand the possibilities and limitations of RTK-SN, which was a significant aid in the work carried out in Paper V. It was also a chance to get familiar with UWB technology, which was necessary to develop the UWB-based position system of the model cars and carry out the experiments presented in Paper VI.

3.5 Paper V: A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving

Here we present our self-driving model vehicle testbed, which is useful for research and development within autonomous driving and accurate positioning. It consists of custom electronics and software developed by us, and can fitted on any model vehicle with Ackerman or differential steering. Figure 6 shows our 1:6 scale model car equipped with the electronics of our testbed.

Our model car uses RTK-SN as an absolute position reference, and fuses this data with odometry feedback from our motor controller as well as measurements from an IMU. This way we can achieve a position update rate of 100 Hz with

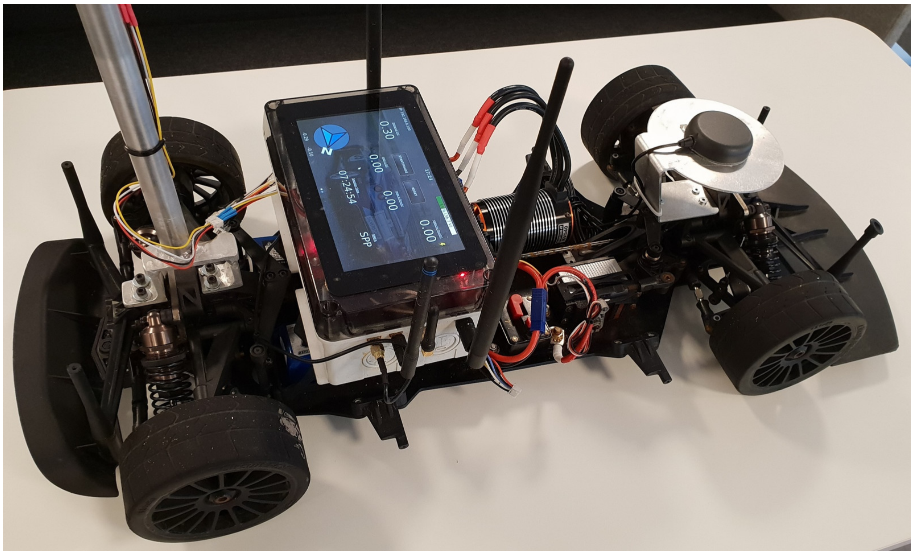


Figure 6: A photo of the 1:6 scale model car in our testbed.

only one sample latency. In comparison, the Ublox M8P[1] RTK-SN receiver we are using outputs position data with 5 Hz where every sample is around 120 ms old. We have verified that our position filter provides an accuracy of around 3 cm under dynamic conditions, as long as there is sufficiently low wheel slip.

In addition to the accurate high rate positioning, our testbed can follow trajectories using the pure pursuit algorithm [34] with some of our improvements. We have also developed a comprehensive Graphical User Interface (GUI) for our testbed where trajectories can be edited on top of OpenStreetMap [35] and a fleet of our model cars can be monitored and controlled in real time.

Our model vehicle testbed can be used for a wide variety of research, development and surveying applications. We use it as a base platform for carrying out the experiments presented in Paper VI.

3.6 Paper VI: Automated Testing of Ultra-Wideband Positioning for Autonomous Driving

In this paper we equip our model car, presented in Paper V, with our UWB positioning modules, presented in Paper IV. Then we update the embedded software of our model car control Printed Circuit Board (PCB) with a sensor fusion filter that derives the position of the car using UWB range measurements betwen the UWB module on the car and several stationary anchors, as shown in Figure 7. This position filter also uses odometry feedback from the motor controller of the car and measurements from the IMU. This way we can derive

[1]https://www.u-blox.com/en/product/neo-m8p-series

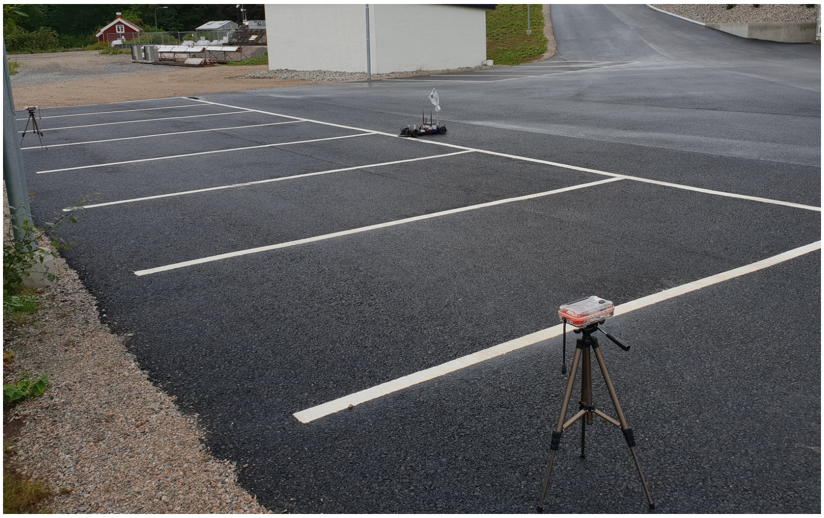


Figure 7: Our model car driving on a parking lot with two UWB anchors mounted on tripods in waterproof boxes.

an absolute position at 100 Hz independent of the RTK-SN measurements, thus providing a redundant position estimate.

Then we use the PBT tool ScalaCheck [10] to generate HIL and full hardware tests for our model car, where we compare the UWB-derived position with the RTK-SN-derived position. While performing tests, we also inject faults using an embedded version of our FaultCheck tool, presented in Paper I, in order to evaluate the fault tolerance of our UWB sensor fusion filter. This is similar to the experiments we performed in Paper II, with the additional challenges of 1) dealing with the real-time nature of the system as opposed to simulated time; 2) being able to reset the system state between tests and 3) maintaining safety during the tests as our automatically generated tests cause our model car to physically drive along randomly generated trajectories. In the paper we go through the details of how we randomly generate drivable trajectories and how we constrain where the car is allowed to drive. We also describe how we generate a trajectory from an arbitrary position and orientation to the start position and orientation of the test scenario in order to generate new tests, or to repeat interesting tests.

To demonstrate the approach, we managed to find several interesting tests where the UWB position deviates more than our defined maximum allowed deviation from the RTK-SN position. We were able to repeat the tests several times, with and without fault injection enabled, with consistent results between consecutive tests.

References

[1] J. P. John, S. S. Kumar, and B. Jaya. ”Space Vector Modulation based Field Oriented Control scheme for Brushless DC motors”. In: *International Conference on Emerging Trends in Electrical and Computer Technology.* Mar. 2011, pp. 346–351.

[2] F. C. Braescu. ”Basic control algorithms for vehicle platooning prototype model car”. In: *21st International Conference on System Theory, Control and Computing (ICSTCC).* Oct. 2017, pp. 180–185.

[3] T. Takasu and A. Yasuda. ”Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB”. In: *International Symposium on GPS/GNSS.* International Convention Centre Jeju, Korea. 2009, pp. 4–6.

[4] M. Skoglund, T. Petig, B. Vedder, H. Eriksson, and E. M. Schiller. ”Static and dynamic performance evaluation of low-cost RTK GPS receivers”. In: *IEEE Intelligent Vehicles Symposium (IV).* June 2016, pp. 16–19.

[5] M. Gaballah and M. El-Bardini. ”Low cost digital signal generation for driving space vector PWM inverter”. In: *Shams Engineering Journal* 4.4 (2013), pp. 763–774.

[6] Decawave. *The implementation of two-way ranging with the DW1000. APS013.* Application Note. Decawave, 2015.

[7] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects.* Ed. by F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 250–271.

[8] R. K. Iyer. ”Experimental Evaluation”. In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing.* FTCS’95. Pasadena, California: IEEE Computer Society, 1995, pp. 115–132.

[9] K. Claessen and J. Hughes. ”QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming.* ICFP ’00. New York, NY, USA: ACM, 2000, pp. 268–279.

[10] R. Nilsson. *ScalaCheck: The Definitive Guide.* Artima Press, 2014.

[11] T. Arts, J. Hughes, J. Johansson, and U. Wiger. ”Testing Telecoms Software with Quviq QuickCheck”. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang.* Portland, Oregon: ACM Press, 2006.

[12] T. Morimoto. *pytest-quickcheck.* 2015. URL: https://pypi.python.org/pypi/pytest-quickcheck/.

[13] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. ”ArbitCheck: A Highly Automated Property-Based Testing Tool for Java”. In: *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* Mar. 2014, pp. 405–412.

[14] J. Hunt. ”Scala Testing”. English. In: *A Beginner’s Guide to Scala, Object Orientation and Functional Programming.* Springer International Publishing, 2014, pp. 365–382.

[15] M. Gupta. *Akka Essentials.* Community experience distilled. Packt Publishing, 2012.

[16] A. Nilsson, L. M. Castro, S. Rivas, and T. Arts. ”Assessing the Effects of Introducing a New Software Development Process: a Methodological Description”. In: *International Journal on Software Tools for Technology Transfer* (2013), pp. 1–16.

[17] R. Svenningsson, R. Johansson, T. Arts, and U. Norell. ”Formal Methods Based Acceptance Testing for AUTOSAR Exchangeability”. In: *SAE Int. Journal of Passenger Cars– Electronic and Electrical Systems* 5.2 (2012).

[18] I. E. Commission. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems.* Norm. 2010.

[19] I. O. for Standardization ISO. *ISO 26262: Road vehicles – Functional safety.* Norm. 2011.

[20] 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.* 1998, pp. 284–293.

[21] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. ”Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms”. In: *IEEE Micro* 14.1 (1994), pp. 8–23.

[22] J. Gaisler. ”A portable and fault-tolerant microprocessor based on the SPARC v8 architecture”. In: *International Conference on Dependable Systems and Networks (DSN).* 2002, pp. 409–415.

[23] H. Madeira, M. Rela, F. Moreira, and J. Silva. ”RIFLE: A General Purpose Pin-Level Fault Injector”. In: *Dependable Computing — EDCC-1.* Ed. by K. Echtle, D. Hammer, and D. Powell. Vol. 852. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 197–216.

[24] 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.* 1994, pp. 66–75.

[25] V. Sieh, O. Tschache, and F. Balbach. ”VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions”. In: *Proceedings of the Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing.* 1997, pp. 32–36.

[26] S. Han, K. Shin, and H. Rosenberg. ”DOCTOR: an Integrated Software Fault Injection Environment for Distributed Real-Time Systems”. In: *Proceedings of the Computer Performance and Dependability Symposium.* 1995, pp. 204–213.

[27] M. Hiller. ”PROPANE: An Environment for Examining the Propagation of Errors in Software”. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM Press, 2002, pp. 81–85.

[28] Trimble. *GPS The First Global Navigation Satellite System*. Trimble Navigation Limited, 2007.

[29] Decawave. *DWM1000 IEEE 802.15.4 UWB Transceiver Module. DWM1000 Datasheet*. Datasheet. Decawave, 2016.

[30] Decawave. *Sources of Error in DW1000 based Two-Way Ranging (TWR) Schemes. APS011*. Application Note. Decawave, 2015.

[31] AUTOSAR. *Specification of SW-C end-to-end communication protection Library*. Specification. 2013-02-20.

[32] J. Eckert, R. German, and F. Dressler. "ALF: An Autonomous Localization Framework for Self-Localization in Indoor Environments". In: *7th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2011)*. Barcelona, Spain: IEEE, 2011, pp. 1–8.

[33] J. Eckert, R. German, and F. Dressler. "On Autonomous Indoor Flights: High-Quality Real-Time Localization Using Low-Cost". In: *IEEE International Conference on Communications (ICC 2012), IEEE Workshop on Wireless Sensor Actor and Actuator Networks (WiSAAN 2012)*. Ottawa, Canada: IEEE, 2012, pp. 7093–7098.

[34] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro. "Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas". In: *IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. Oct. 2016, pp. 7–12.

[35] M. Haklay and P. Weber. "OpenStreetMap: User-Generated Street Maps". In: *IEEE Pervasive Computing* 7.4 (Oct. 2008), pp. 12–18.

Paper I

Combining Fault-Injection with Property-Based Testing

Benjamin Vedder, Thomas Arts, Jonny Vinter and Magnus Jonsson

Published in Proceedings of the International Workshop on Engineering Simulations for Cyber-Physical Systems (ES4CPS '14), Dresden, Germany, 2014

Abstract

In this paper we present a methodology and a platform using Fault Injection (FI) and Property-Based Testing (PBT). PBT is a technique in which test cases are automatically generated from a specification of a system property. The generated test cases vary input stimuli as well as the sequence in which commands are executed. FI is used to accelerate the occurrences of faults in a system to exercise and evaluate fault handling mechanisms and e.g. calculate error detection coverage. By combining the two we have achieved a way of randomly injecting different faults at arbitrary moments in the execution sequence while checking whether certain properties still hold. We use the commercially available tool QuickCheck for generating the test cases and developed FaultCheck for FI. FaultCheck enables the user to utilize fault models, commonly used during FI, from PBT tools like QuickCheck. We demonstrate our method and tools on a simplified example of two Airbag systems that should meet safety requirements. We can easily find a safety violation in one of the examples, whereas by using the AUTOSAR E2E-library implementation, exhaustive testing cannot reveal any such safety violation. This demonstrates that our approach on testing can reveal certain safety violations in a cost-effective way.

1 Introduction

Testing cannot reveal the absence of software defects, but it is one of the most cost effective ways of demonstrating that certain requirements are not fulfilled. Property-Based Testing (PBT) with QuickCheck [1] is a demonstrated way to get more effective testing in a cost effective way [2]. Among others, it has been used for large scale testing of AUTOSAR software [3]. When using PBT for complex software, a model is created that describes certain properties of the software specified by so called *functional requirements*. The PBT tool automatically generates and runs many tests in an attempt to falsify these properties. When the properties are falsified a counterexample in a minimalistic form is shown that can be used to either fix a bug in the implementation or to revise the specification. The case where the PBT tool is not able to falsify a property is not a formal proof that the property holds, but it shows that it holds for numerous randomly generated inputs.

In particular when it comes to costly procedures of certifying software where additional arguments have to be provided concerning the correctness of the implementation, one would better be sure it is very well tested. Not only for the common case, but also in cases in which faults occur and the software should deal with those faults.

Under normal circumstances it is unlikely that errors occur that the software should react upon. Therefore the purpose of Fault Injection (FI) is to introduce errors while testing. Thus, the goal of FI is to inject faults into software and/or the hardware connected to the software in order to ensure that the system still fulfils certain requirements while these faults are present. For example, in safety-critical systems, it has to be made sure that the system is not dangerous when certain faults are present. This means that FI deals with *non-functional requirements*.

In general, the models used for PBT are describing the behaviour of the system in case no faults occur. Injecting faults might very well change the functional behaviour and make tests fail w.r.t. the functional behaviour, even though this behaviour would still be correct from the safety point of view. The challenge we address in this paper is to enhance the model with specifying its behaviour in case of occurring faults, but still be able to detect functional defects in case no faults occur. For that, we need to make the models aware of the injection of faults and the generated test cases should also be controlling the fault injection.

In this paper, we present a method for combining PBT and FI in order to test safety-critical systems in an effective way. According to our knowledge, this has not been reported before. We demonstrate the gained possibilities through experiments with an example utilizing the AUTOSAR End-to-End (E2E)-library.

The rest of the paper is organized as follows. In Section 3.1 we introduce FI with related work, and in Section 3 we introduce PBT with related work. In Section 4 we introduce the FaultCheck tool that is developed within this study. Section 5 shows a use case where this platform is used and in Section 6 we present our conclusions from this study.

2 Fault Injection

FI is used to accelerate the occurrences of faults in a system [4]. The aim is to exercise and evaluate fault handling mechanisms and calculate fault-tolerance measures such as error detection coverage. Traditionally, targets for FI have been hardware (microprocessors and memories), simulation models of hardware and software running on hardware. Examples on different approaches for hardware-implemented FI includes heavy-ion FI [5], pin-level FI [6, 7], Scan-chain implemented FI [8] and Nexus-based FI [9]. Examples on approaches for FI in models of hardware include simulation-based FI [10–12]. Software-implemented FI can be used pre-runtime [13] or during runtime [14]. Other approaches have been presented in the literature dealing with FI directly in source code [15] and in models of, e.g. software (denoted here as model-based FI) [16, 17] from which source code may be generated. Such techniques can be used early in the software development process and dependability flaws can be corrected less costly compared to flaws found during later phases. Model-based FI is particularly useful during model-based development of systems. This paper presents ideas on how to combine knowledge from FI (fault tolerance against hardware faults) with knowledge on PBT (fault removal of software faults) to enhance PBT testing tools such as QuickCheck. The ideas are implemented in a tool called FaultCheck, which is presented in detail in Section 4.

One simple example of model-based fault injection is illustrated in Figure 1. In this example, a model is fed with the same inputs over and over again while different faults are injected. The output is compared to the output from

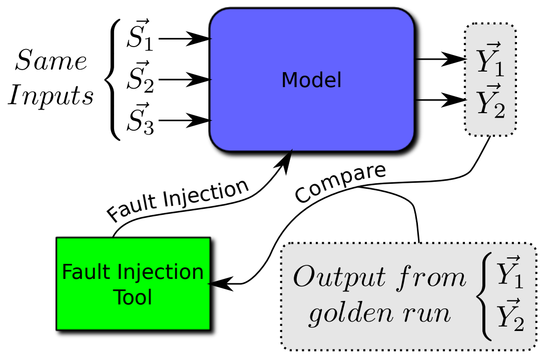


Figure 1: One example of model-based fault injection.

a *golden run* where everything was run without faults present.

In summary, fault injection deals with showing how fault tolerant a system is, evaluating fault handling mechanisms and determining error detection coverage. One opportunity for improvement lies in checking how numerous auto-generated input sequences affect the impact of faults.

3 Property-Based Testing

Property-based testing [18] is a technique in which test cases are automatically generated from a specified property of a system. For example, if we would have a function computing one trusted value from three possibly different sensor values, then one could express a property of the output value in relation to the input values. With PBT one would automatically generate test sequences that vary the sensor input data and compare the output data with the "modelled" output data in the property.

One example of PBT is shown in Figure 2 where the QuickCheck tool [1, 19] is used. In this example, a very simple application is tested and the model contains all the details of the implementation. In more sophisticated applications, the model might contain only a few details of the implementation relevant for evaluating the interesting properties.

3.1 Combing Property-Based Testing and Fault Injection

In this paper we present how techniques from PBT and FI are used together. Advantages are that we can automatically generate test cases and check if the state of the System Under Test (SUT) is as expected while injecting faults at the same time.

In order to combine FI with PBT, a set-up as illustrated in Figure 3 is used. We show that it is possible to define properties that are supposed to hold with certain faults present in the system and run tests with randomly generated Application Programming Interface (API) calls.

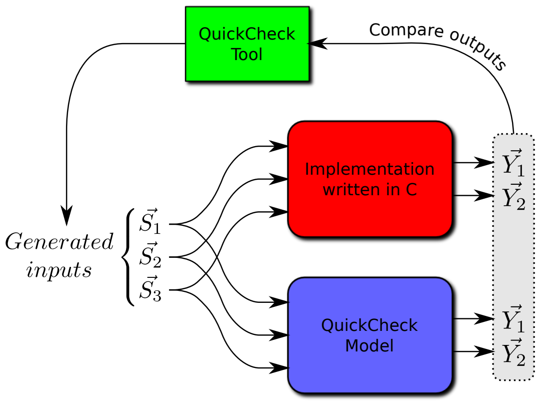


Figure 2: One example of PBT.

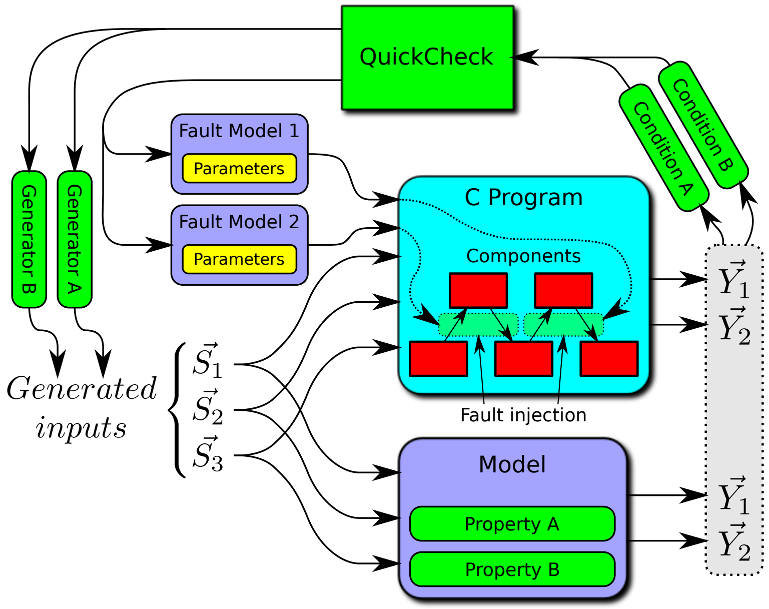


Figure 3: Property-based testing combined with fault injection inside a C program.

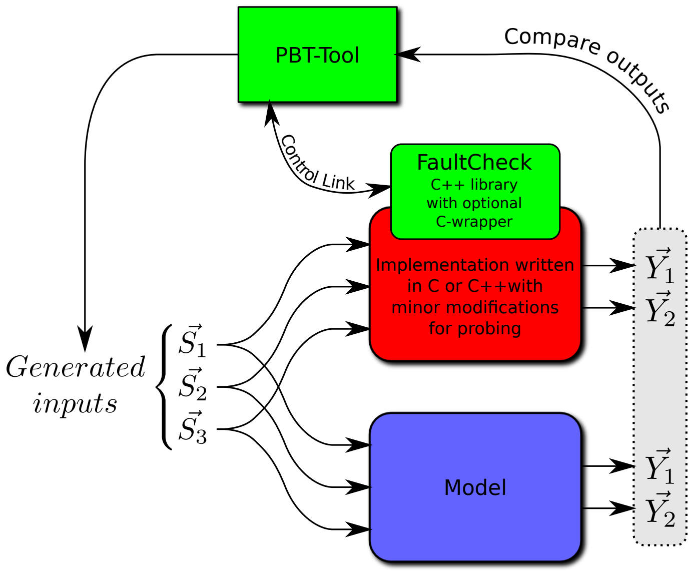


Figure 4: The FaultCheck probing-library controlled from a PBT-tool.

In this study we have implemented one way to perform FI on C code while using techniques from PBT and QuickCheck to generate thousands of "golden runs" automatically. The approach presented in this paper shows the concept of the idea.

4 FaultCheck

FaultCheck is a tool under early development with the aim to support FI into C and C++-code. It consists of a library written in C++ with a wrapper around C, so it can easily be included and linked against in existing applications. FaultCheck is designed to be used by other tools that perform property-based testing with QuickCheck or other PBT tools, such as ScalaCheck [20]. A block diagram of a typical use case of FaultCheck is shown in Figure 4.

Currently, there are two parts of FaultCheck under development:

Probing, which is done by modifying existing C and C++ applications. This way, faults can be injected anywhere in the application, to simulate hardware faults that manifests as errors at the software level, at the cost of some overhead in execution time and code space.

Communication channel emulation, which is an interface that provides an emulated communication channel into which a number of communication faults can be injected to evaluate to which extent the application can handle them.

The motivation for making the communication channel accessible from C is that many programming languages such as Python [21], Java and Scala [22] have the ability to interface with C libraries. In this way, FaultCheck can easily be used from other tools and languages.

4.1 Fault Models

A hardware fault model can be defined as the number of faults, the duration and the type of the fault. An example is a single transient bit-flip fault. Another example is multiple permanent stuck-at-zero faults. In this study we have implemented several hardware fault models as well as communication fault models. Several of these fault models are handled by the AUTOSAR E2E library [23]. We have tested that injecting these faults when using that library did not violate our safety requirements.

4.2 Supported Fault Models

FaultCheck currently supports the following fault models that can be injected into C and C++-code via probing:

- BITFLIP
  - Flips a specific bit in a variable.
- BITFLIP_RANDOM
  - Flips a randomly selected bit in a variable.
- STUCK_TO_CURRENT
  - Freezes a variable to the last value it had.
- SET_TO
  - Sets a variable to a pre-set value.
- AMPLIFICATION
  - Scales a variable with a factor.
- OFFSET
  - Adds an offset to a variable.

The communication channel emulation currently supports these fault models:

- REPEAT
  - Repeats a packet a number of times.
- DROP
  - Drops a number of packets (loss of information).
- CORRUPTION

– Alters the data of a packet with any of the fault models specified in the probing part for variables. Since the same code as for the probing interface is used, features added to the probing interface can be made available for the corruption fault easily.

4.3 Probing C-Code

The probing interface of FaultCheck can be used by including the FaultCheck headers and linking against its dynamic libraries. This way of probing C code has been inspired by a tool called PROPANE [24] which supports fault injection probes and monitoring probes.

The following sample code shows how a C application can be probed by using the FaultCheck tool:

```
#include "faultcheck_wrapper.h"

// C code...

SensorValue sensor_evaluate(int S1val, int S2val, int S3val) {

    SensorValue sv;

    // some code...

    faultcheck_injectFaultInt("S1val", &S1val);
    faultcheck_injectFaultInt("S2val", &S2val);
    faultcheck_injectFaultInt("S3val", &S3val);

    // Some more code...

    return sv;
}
```

Here, pointers to the integers S1val, S2val and S3val are sent to FaultCheck with string identifiers, and based on the configuration and previous events they may be modified.

Probing combined with the triggering functionality described in Section 4.6 can, for example, be used to precisely affect certain iterations of loops in C programs in a way that is not possible by only using the external interface of the program.

4.4 Communication Channel Emulation

A packet-based communication channel can be emulated by FaultCheck and used from C programs. The following example shows how one packet is encapsulated by the AUTOSAR E2E-library [23] and passed to the communication channel emulated by FaultCheck.

```
void sensor(unsigned char *data) {
  unsigned char buffer[config.DataLength + 2];

  memcpy(buffer + 1, data, config.DataLength);

  E2E_P01Protect(&config, &sender_state, buffer);
  faultcheck_packet_addPacket("airbag",
    (char*)buffer, config.DataLength + 2);
}
```

This packet will be added to a buffer in FaultCheck and can later be read by using *faultcheck_packet_getPacket* in a similar manner from the application. When the packet is read, the communication channel faults that are enabled will be applied to the packet.

4.5 Integration with other Tools

FaultCheck is not intended to be used as a stand-alone testing framework. It should be used together with other tools that preferably use PBT. Many tools that do PBT on C code already have access to the interface of that code, therefore FaultCheck extends the interface of the C applications with functions to control the FI. This means that the tool that performs PBT can use the functions provided by FaultCheck in the same way it uses the other functions of that C application.

The following sample code shows how to activate a bit-flip fault for one identifier:

```
faultcheck_addFaultBitFlip("S1val", 3, 1);
```

This would flip bit number three (zero is the least significant bit) in the variable *S1val* in the sample shown in Section 4.3. Every time the function *sensor_evaluate* would be called after this point, the third bit of *S1val* would be flipped.

Multiple faults can also be added to the same identifier at the same time. They will be applied to the data in the order they were added. For example, in order to first flip bit number two, then add 23 and then flip bit number 11, the following calls would be used:

```
faultcheck_addFaultBitFlip("S1val", 2);
faultcheck_addFaultOffset("S1val", 23);
faultcheck_addFaultBitFlip("S1val", 11);
```

Note that this combination of faults might not be a realistic fault model, but it shows the flexibility of FaultCheck to design complex fault models.

4.6 Temporal Triggers for Faults

In addition to information about the type and parameters of the faults, FaultCheck also keeps track of when faults should be activated. This mechanism is called temporal triggering and essentially means that fault activations can be delayed for a number of iterations and then be active for a number of iterations. The notion of iteration in this context means that every time the probing function is called, one iteration has occurred. As multiple faults can be added to each identifier, each of these faults can also have an individual trigger.

The following is an example where multiple faults are added for the same identifier (*S1val*) with different triggers:

```
faultcheck_addFaultBitFlip("S1val", 2);
faultcheck_setTriggerAfterIterations("S1val", 120);
faultcheck_setDurationAfterTrigger("S1val", 45);

faultcheck_addFaultOffset("S1val", 23);
faultcheck_setTriggerAfterIterations("S1val", 45);
faultcheck_setDurationAfterTrigger("S1val", 200);

faultcheck_addFaultBitFlip("S1val", 11);
faultcheck_setTriggerAfterIterations("S1val", 130);
faultcheck_setDurationAfterTrigger("S1val", 10);
```

The trigger will be applied to the latest fault that was added. So, the way to add multiple faults with triggers is to add one fault, set the trigger for it, add another fault, set the trigger for it and so on. The code snippet above will cause the following to happen:

- A bit flip on bit number two that is active from iteration 120 and for 45 iterations after that.
- An offset of 23 that is active from iteration 45 and for 200 iterations after that.
- A bit flip on bit number 11 that is active from iteration 130 and for 10 iterations after that.

Note that the offset will be triggered before the first bit flip, but when the first bit flip is also triggered it will be applied before the offset. This scenario will occur during iteration 120 to iteration 165. When several faults are triggered at the same time, they will be applied to the variable in the same order as they were added in the code.

The previous examples illustrate how to use triggers with the probing functionality, but triggers can also be used in the same way with the communication channel of FaultCheck. For the communication channel, one iteration is defined as every time *faultcheck_packet_getPacket* is called.

5 An example: AUTOSAR E2E

When designing safety-critical control systems, one needs to ensure that safety requirements are met. In different domain areas corresponding safety standards help to guide safety engineers through the process of formulating requirements and designing for safety.

In the automotive domain the standard that covers functional safety aspects of the entire development process is called ISO 26262 [25], which is an adaptation of the IEC 61508 [26] standard. The AUTOSAR E2E-library takes the ISO 26262-standard into consideration and is supposed to work with all Automotive Safety Integrity Levels (ASILs) regarding communication when the implementation recommendation is followed. This can reduce the development effort and make the implementation compatible with other AUTOSAR components.

In short, the AUTOSAR E2E library supports the detection of corruption and loss of data when transporting data from one end to the other. This is a building block for safety solutions, since typical fault models include that data on a bus may occasionally be corrupt or even be totally lost.

On top of this library we developed an example of an airbag system with three sensors to detect a collision. The sensors are continuously sampled and their combined data is constantly sent to the airbag ignition system. It would be unsafe if that data could be corrupted in such a way that the airbag would spontaneously fire. Hence, the airbag ignition system needs to know that the data received from the sensors can be trusted. Thus, safety requirements state that the airbag should explode when the car crashes, but even more important that it does not explode at low speed or without a crash.

5.1 Experiment Set-up

In order to test to whether the E2E-library indeed offers good protection against some of the fault models it is designed to handle, a system as shown in Figure 5 has been developed. It works in the following manner:

1. A series of calls is generated and fed into a C application that uses the AUTOSAR E2E-library. In this case, a series of calls with a certain command, $[85, 170]$, should update a variable in the state. When these commands are not sent, the state is not allowed to change regardless of what other commands are sent.

2. The commands from the calls are encapsulated by the E2E protection wrapper that uses the E2E library. The encapsulated packets are then passed to the FaultCheck tool.

3. If a fault is activated, FaultCheck will alter the packet based on the chosen fault model before it is fetched by the application.

4. The E2E protection wrapper fetches packets from FaultCheck and checks them by using the E2E library. Then it performs state updates based on the results.

5. The QuickCheck tool analyses the state of the application and determines whether the state follows the specification or not.

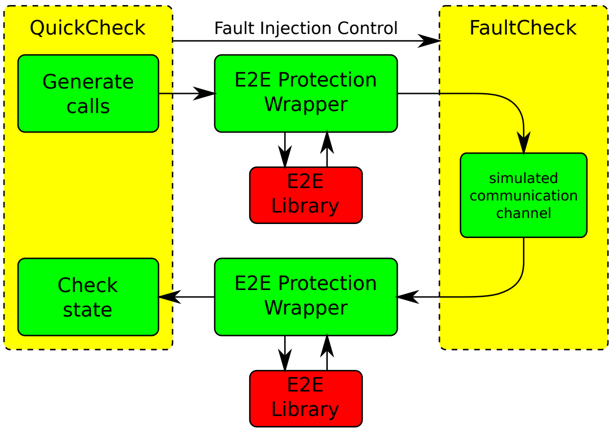


Figure 5: Experiment set-up of the E2E evaluation system.

The reason that multiple commands are sent to update the state is that the AUTOSAR E2E-library recommends that the application is able to handle one faulty packet by itself.

In order to validate our framework, we also tested the application without using the AUTOSAR E2E-library to see which failures we can detect. Later we confirmed that the E2E-library protects against the faults that cause these failures.

The C-code is tested with four different QuickCheck-commands: *sensor*, *bit_flip*, *repetition*, and *explosion*. The *sensor* command is run 10 time more frequently than the other commands and looks as follows:

```
sensor(Data) ->
  DataPtr = eqc_c:create_array(unsigned_char,Data),
  case ?USE_E2E of
    yes ->
      c_call:sensor_e2e(DataPtr),
      c_call:airbag_iteration_e2e();
    no ->
      c_call:sensor(DataPtr),
      c_call:airbag_iteration()
  end,
  eqc_c:value_of(airbag_active).

% Always send 170 as the second byte, otherwise it is
% very unlikely that an explosion will occur since a
% double fault would be required most of the time.
sensor_args(_S) ->
  [[byte(), 170]].

sensor_pre(_S, [Data]) ->
```

```
  not is_explode(Data).

sensor_post(_S, [_Data], Res) ->
  Res == 0.
```

This command will generate a combination of two bytes that should *not* fire the airbag (e.g. not $[85, 170]$) and send them to the application. The postcondition is that the airbag should not fire, and if this is not true (the airbag state is set to fired) the property will evaluate as false.

The *bit_flip* command is used to make FaultCheck inject a *CORRUPTION* fault of the bit flip type into the communication channel. It looks in the following way:

```
bit_flip(Byte,Bit) ->
    c_call:faultcheck_packet_addFaultCorruptionBitFlip("airbag", Byte
, Bit).

bit_flip_args(_S) ->
    case ?USE_E2E of
      yes -> [ choose(0,3), choose(0,7) ];
      no -> [ choose(0,1), choose(0,7) ]
    end.

bit_flip_pre(S,[Byte,Bit]) ->
    length(S#state.faults) < 4 andalso
  not lists:member({bitflip,Byte,Bit},S#state.faults).

bit_flip_next(S, _, [Byte, Bit]) ->
    S#state{faults = S#state.faults ++ [{bitflip, Byte, Bit}]}.
```

The arguments are the byte to affect in the packet (0 to 1 without the E2E-library or 0 to 3 with the E2E-library) and which bit to flip in that byte (0 to 7). The precondition to run this command is that there is not already another bit flip on the same bit, as this would be meaningless because they take each other out. Another part of the precondition is that there are less than 4 faults active simultaneously.

The *repetition* command will make FaultCheck repeat a packet a certain number of times. It looks as follows:

```
repetition(Num) ->
    c_call:faultcheck_packet_addFaultRepeat("airbag", Num).

repetition_args(_S) ->
    [ choose(1, 3) ].

repetition_pre(S, [Num]) ->
    length(S#state.faults) < 4 andalso
  not lists:member({repetition, Num}, S#state.faults).

```

```
repetition_next(S, _, [Num]) ->
    S#state{faults = S#state.faults ++ [{repetition, Num}]}.
```

The *explosion* command is used to test that the airbag actually will explode when there are no faults. It looks as follows:

```
explosion(Data) ->
  DataPtr = eqc_c:create_array(unsigned_char, Data),
  c_call:faultcheck_packet_removeAllFaults(),
  case ?USE_E2E of
    yes ->
      % Call the sensor function often enough for the E2E library
      % to recover (15 times)
      [ c_call:sensor_e2e(DataPtr) || _<-lists:seq(1,15) ],
      c_call:airbag_iteration_e2e();
    no ->
      c_call:sensor(DataPtr),
      c_call:sensor(DataPtr),
      c_call:airbag_iteration()
  end,
  Res = eqc_c:value_of(airbag_active),
  c_call:application_init(),
  Res.

% The arguments for the explosion command are always [85, 170]
explosion_args(_S) ->
  [[85, 170]].

% The postcondition is that the airbag should explode
explosion_post(_S, [_Data], Res) ->
  Res == 1.
```

The reason that the explosion command calls the sensor function several times is to give the E2E-library a chance to recover after many possible injected faults. The reason to use this command is to make sure that the E2E-library actually will pass data to the application when there is no fault present.

5.2 Experiment Results

First we tested the application without the E2E-library to see how it behaves. The following commands constitute a typical sequence that makes the airbag explode when it should not:

```
airbag_eqc:bit_flip(0, 1) -> ok
airbag_eqc:repetition(1) -> ok
airbag_eqc:sensor([87, 170]) -> 1
```

This sequence shows one command that flips bit number 1 in byte number 0. The next command will repeat whatever is sent once. The third command, *sensor*, will send [87, 170] to the airbag. 87 will be changed to 85 when the first bit is flipped and then [85, 170] will be sent twice because of the repetition.

Note that this short sequence is easy to understand and exactly points to the problem. The small sequence is obtained from a much longer sequence of calls that failed. QuickCheck automatically searches for smaller failing test cases when a failure is detected. Thus, all commands unnecessary for this unintended explosion to occur are removed by QuickCheck's shrinking technique. As two consecutive commands are required for the airbag to explode, the repetition fault combined with the bit flip were necessary.

It took around 1000 auto-generated tests before this unintended explosion occurred, even when one out of 10 commands was an injected fault, so the mechanism in the application to require two consecutive commands was useful. Without that mechanism, when only one [85, 170] command for an explosion was required, it usually took less than 50 tests with the otherwise same set-up for the failure to occur.

When we run the same QuickCheck model against the Airbag implementation based upon the AUTOSAR E2E-library, the airbag never exploded unintentionally. Not even after running more than 100 000 tests.

Since the possible combinations of injected faults and state of the system is huge, one cannot draw much conclusion from a lot of passing test cases. Failing test cases reveal a problem, but until you find that, you cannot say much about the implementation. The distribution of the test data, collected during testing, is the only hint we have to see what we tested and whether we think this is a good test distribution.

While analysing the data, we realized that it is hard to jump from an arbitrary number to 85 by flipping a bit. We did reduce the search space by instead of sending two arbitrary integers, always send 170 as the second integer and choose the first integer in the set of values {84, 87, 117, 213, 21}, i.e., values that easily mutate to 85. The data generator to express this is written in QuickCheck as follows:

```
sensor_args(_S) ->
  [[elements([84, 87, 117, 213, 21]), 170]].
```

Here only one bit differs an innocent command from a command that causes an explosion. This made the application fail after less than 50 tests most of the time without the AUTOSAR E2E-library (as opposed to 1000 tests). The test output typically looked like the following after shrinking:

```
airbag_eqc:bit_flip(0, 0) -> ok
airbag_eqc:sensor([84, 170]) -> 0
airbag_eqc:sensor([84, 170]) -> 1
```

The first command flips bit number 0 in byte number 0. This will cause [84, 170] to be changed to [85, 170] and two such commands will make the airbag explode. In this case, two *sensor* commands with the same data were more likely to occur than a *repetition* command because of the limited amount of data for the sensor command to be chosen from.

However, with the E2E-library included, even 100 000 tests with the modified generator would not make the airbag explode when it should not, while the explosion command that sends [85, 170] still could make the airbag explode when the faults where disabled for several iterations.

In this experiment, the *Protection Wrapper* communicates directly with FaultCheck and is not the one provided by the real application that uses one of the communication interfaces provided by AUTOSAR. This does however not imply that the Protection Wrapper of the application to be tested has to be modified to support the FaultCheck interface. One might as well use an existing Protection Wrapper and mock the interface that AUTOSAR provides with an additional C component. This way, the same experiment can be carried out without intruding on the original Protection Wrapper of the application. The only reason that we connected the Protection Wrapper directly to FaultCheck in the example here is that we do not have any different protection wrapper to begin with; and therefore we could as well connect the one we create directly to FaultCheck.

6 Conclusions

We have presented a platform and methodology that uses FI and PBT to test safety-critical systems. Advantages include that faults can be injected while inputs are auto-generated based on properties and property-based checks on the software under investigation are performed.

An Airbag example based on the AUTOSAR E2E-library is presented where we run thousands of auto-generated tests. In this experiment, we have discovered faults that will cause unexpected behaviour with certain inputs when the E2E-library is disabled. We have also confirmed that enabling the E2E library will protect against these types of faults. This way, we have shown how non-functional requirements can be tested by using FI combined with PBT.

This methodology presents how to combine FI with PBT for evaluation of realistic use cases with one or more software components in order to exercise and evaluate fault handling mechanisms. This will indicate whether the evaluated fault handling mechanism is enough to cope with the expected faults or if something additional is needed.

Although the AUTOSAR example is from the automotive industry, which is one of the areas where fault injection is used today, the same techniques can be applied to other areas. Wherever it makes sense to test one or several parts of a system in a realistic use case, this platform can be used to evaluate fault handling capabilities.

7 Acknowledgement

This research has been funded by the PROWESS EU project (Grant agreement no: 317820) [27]. We would like to thank Hans Svensson for useful feedback and comments.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. ”Testing Telecoms Software with Quviq QuickCheck”. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang*. Portland, Oregon: ACM Press, 2006.

[2] A. Nilsson, L. M. Castro, S. Rivas, and T. Arts. ”Assessing the Effects of Introducing a New Software Development Process: a Methodological Description”. In: *International Journal on Software Tools for Technology Transfer* (2013), pp. 1–16.

[3] R. Svenningsson, R. Johansson, T. Arts, and U. Norell. ”Formal Methods Based Acceptance Testing for AUTOSAR Exchangeability”. In: *SAE Int. Journal of Passenger Cars– Electronic and Electrical Systems* 5.2 (2012).

[4] R. K. Iyer. ”Experimental Evaluation”. In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing*. FTCS’95. Pasadena, California: IEEE Computer Society, 1995, pp. 115–132.

[5] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. ”Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms”. In: *IEEE Micro* 14.1 (1994), pp. 8–23.

[6] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell. ”Fault Injection for Dependability Validation: A Methodology and Some Applications”. In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 166–182.

[7] H. Madeira, M. Rela, F. Moreira, and J. Silva. ”RIFLE: A General Purpose Pin-Level Fault Injector”. In: *Dependable Computing — EDCC-1*. Ed. by K. Echtle, D. Hammer, and D. Powell. Vol. 852. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 197–216.

[8] 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*. 1998, pp. 284–293.

[9] 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*. 2001, pp. 83–88.

[10] 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*. 1994, pp. 66–75.

[11] V. Sieh, O. Tschache, and F. Balbach. ”VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions”. In: *Proceedings of the Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*. 1997, pp. 32–36.

[12] K. K. Goswami, R. K. Iyer, and L. Young. ”DEPEND: A Simulation-Based Environment for System Level Dependability Analysis”. In: *IEEE Transactions on Computers* 46 (1997), pp. 60–74.

[13] S. Han, K. Shin, and H. Rosenberg. ”DOCTOR: an Integrated Software Fault Injection Environment for Distributed Real-Time Systems”. In: *Proceedings of the Computer Performance and Dependability Symposium*. 1995, pp. 204–213.

[14] J. Carreira, H. Madeira, J. G. S., and D. E. Informática. ”Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers”. In: *IEEE Transactions on Software Engineering* 24 (1998), pp. 125–136.

[15] M. Hiller. ”A Software Profiling Methodology for Design and Assessment of Dependable Software”. Ph.D Thesis. Göteborg: Chalmers University of Technology, 2002.

[16] 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*. 2007, pp. 1–9.

[17] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren. ”MODIFI: a Model-Implemented Fault Injection tool”. In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*. SAFECOMP'10. Vienna, Austria: Springer-Verlag, 2010, pp. 210–222.

[18] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects*. Ed. by F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 250–271.

[19] K. Claessen and J. Hughes. ”QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. 2000, pp. 268–279.

[20] R. Nilsson. *ScalaCheck*. 2013. URL: https://github.com/rickynils/scalacheck.

[21] *Python/C API reference manual*. 2013. URL: http://docs.python.org/2/c-api/.

[22] *bridj*. 2013. URL: http://code.google.com/p/bridj/.

[23] AUTOSAR. *Specification of SW-C end-to-end communication protection Library*. Specification. 2013-02-20.

[24] M. Hiller. ”PROPANE: An Environment for Examining the Propagation of Errors in Software”. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 2002, pp. 81–85.

[25] I. O. for Standardization ISO. *ISO 26262: Road vehicles – Functional safety*. Norm. 2011.

[26] I. E. Commission. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems*. Norm. 2010.

[27] P. Consortium. *Property-Based Testing for Web Services*. 2014. URL: http://www.prowessproject.eu/.

Paper II

Towards Collision Avoidance for Commodity Hardware Quadcopters with Ultrasound Localization

Benjamin Vedder, Henrik Eriksson, Daniel Skarin, Jonny Vinter and Magnus Jonsson

Published in Proceedings of the International Conference on Unmanned Aircraft Systems (ICUAS), Denver, Colorado, USA, (2015).

Abstract

We present a quadcopter platform built with commodity hardware that is able to do localization in GNSS-denied areas and avoid collisions by using a novel easy-to-setup and inexpensive ultrasound-localization system. We address the challenge to accurately estimate the copter's position and not hit any obstacles, including other, moving, quadcopters. The quadcopters avoid collisions by placing contours that represent risk around static and dynamic objects and acting if the risk contours overlap with ones own comfort zone. Position and velocity information is communicated between the copters to make them aware of each other. The shape and size of the risk contours are continuously updated based on the relative speed and distance to the obstacles and the current estimated localization accuracy. Thus, the collision-avoidance system is autonomous and only interferes with human or machine control of the quadcopter if the situation is hazardous. In the development of this platform we used our own simulation system using fault-injection (sensor faults, communication faults) together with automatically-generated tests to identify problematic scenarios for which the localization and risk contour parameters had to be adjusted. In the end, we were able to run thousands of simulations without any collisions, giving us confidence that also many real quadcopters can manoeuvre collision free in space-constrained GNSS-denied areas.

1 Introduction

In order to test and demonstrate different applications on Micro Air Vehicles (MAVs), a platform that is easy to set-up and safe to operate can be very useful. We envision a quadcopter platform built from inexpensive hardware that can be set up at new locations in less than 15 minutes. Our targeted environments have constraints on space and lack of Global Navigation Satellite Systems (GNSSs), which makes it difficult to navigate autonomously compared to outdoor environments. This platform should give the pilot, who can be a human or a machine, full control in normal circumstances while preventing collisions when the situation gets hazardous, regardless of pilot input. Thus, the quadcopters have to be aware of their own positions and the positions of static and moving objects in the area. They also have to be aware of the accuracy of their position and the physics that restrict how they can manoeuvre.

Our platform is designed to meet the following requirements:

- No sensitivity to lighting conditions and background contrast, as is the case with camera-based systems [1–4].
- The computations for estimating the position and avoiding collisions should be inexpensive enough to be handled by on-board microcontrollers (as opposed to offloading them to external computers [3, 4]).
- The extra equipment on each quadcopter should be light enough to allow extra payload and spare the battery. There are solutions with relatively heavy laser range finders that do not meet this requirement [5, 6].

- There should be fault tolerance to e.g. handle occasional faulty distance measurements.

- Pilot errors should be handled by automatically taking over control if the situation becomes hazardous.

To meet these requirements, we have created a localization system that uses ultrasound to measure the distance between the copters and several stationary anchors. The ultrasound-localization hardware is based on open-source radio boards [7]. To make the copter's aware of each other, they communicate their positions and velocities to each other on a regular basis.

Testing the system has been a significant part of this work. We have developed a simulator that operates together with fault injection [8] and property-based testing [9] techniques to evaluate how a larger system with quadcopters behaves while hardware faults and/or pilot misbehaviour occurs. This way, we could randomly generate pilot control commands and inject faults during thousands of automatically generated simulations to see when a collision occurs. For fault injection, we used the FaultCheck tool [10] and for generating tests we used the Erlang QuickCheck tool [11]. When we had a sequence of pilot and fault injection commands that led to a collision, we used the shrinking feature of QuickCheck to get a shorter test sequence of commands that leads to a collision. We could then run this sequence of commands in the simulator repeatedly, while adjusting the system parameters, until it would not lead to a collision anymore.

Dealing with the slow update rate of the anchors, with the simulation-hardware relation, and with the occasional measurement faults of the system was challenging. Even so, we achieved a result with a functioning copter platform and much shared code between the simulator and the hardware.

The contributions of this work are the following:

- A novel hardware and software solution for doing localization in GNSS-denied areas based on ultrasound measurements fused with Inertial Measurement Unit (IMU) data using easily available, inexpensive hardware.

- A technique to take over control in hazardous situations to avoid collisions between moving quadcopters by using communication between them and risk contours.

- We show how performance and fault tolerance can be evaluated with automatically generated tests using our previously proposed platform that utilizes fault injection and property-based testing [10].

The rest of the paper is organized as follows. Section 2 presents related research, Section 3 describes our hardware platform, Section 4 describes our ultrasound distance measurement technique and Section 5 shows how we do position estimation. Further, in Section 6 we describe our collision-avoidance technique, Section 7 describes our simulations and in Section 5 we present our conclusions from this work.

2 Related Work

Much research has been devoted to autonomous MAVs, such as quadcopters. Early systems worked only outdoors as they relied on GNSS positioning systems [12]. Recently, part of this research has been devoted platforms that operate in GPS-denied areas such as indoor environments [1–6, 13, 14]. One approach is to use cameras either mounted on the copters to identify the environment [2, 4, 13]; or external cameras that identify markers on the copters [1, 3]. Limitations with the camera-based solutions are that they require much computational power and good light/contrast conditions. Many camera-based solutions run the computation on a stationary computer and send the results back to the copter [1–3, 13, 14]. Another approach is to use laser range finders mounted on the copters to run Simultaneous Localization and Mapping (SLAM) algorithms [5, 6]. This approach often works without modifying the external environment with e.g. anchors or cameras, but relies on the environment having walls that are close enough to be detected. Limitations with laser range finders are that they are relatively expensive and quite heavy, adding much payload to the weight-constrained copter.

Similar to our platform, there is one early system that relies on infra-red and ultrasound sensors mounted on quadcopters that measure distances to walls and the floor [15]. These copters can avoid collisions, but did not have enough accuracy to perform a stable hover. More recently, a platform has been presented by J. Eckert that uses ultrasound localization with inexpensive hardware to manoeuvre quadcopters [14, 16, 17]. This platform uses a swarm of small robots that spread out on the floor and allow a consumer (a quadcopter in this case) to hover above them. Compared to our platform, Eckerts's ultrasound system has a shorter range, of 2 m when there is noise from quadcopters, while our system can operate at distance of up to 12 m from the anchors with the current configuration. Eckert's quadcopter also relies on optical flow sensors aimed towards the floor and ceiling because the update rate from their ultrasound system is too low and not as tightly coupled to the control loop as our system. Thus, their localization depends on having relatively good contrast and lightening conditions and a ceiling that is low enough, which makes it difficult to use outdoors.

To our knowledge, beside our quadcopter system, there is currently no other indoor quadcopter system that can do a stable hover and collision avoidance with only ultrasound localization and IMU-based dead reckoning. Our system also has a unique approach on collision avoidance and fault tolerance.

3 Hardware Setup

Our platform consists of several quadcopters and several (at least two) stationary anchors, as shown in Figure 1. The anchors and quadcopters have synchronized clocks to do Time of Flight (ToF) measurement of ultrasound to determine the distance between them. The copters also have one ultrasound sensor each that measures the distance to the floor. Since the $[x, y, z]^T$ position of the anchors

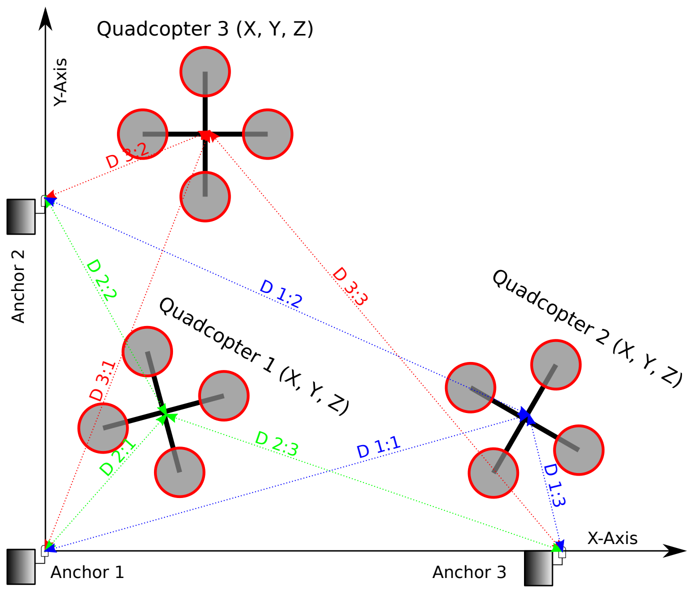


Figure 1: The ultrasound-localization system with anchors. The measured distances from the copters to the anchors are marked D a:b.

is known by the copters, they can calculate their own position based on the distances to the anchors. Two anchors are enough for this system to work if the copters never pass the line between the anchors, but any number of anchors can be used to provide more accuracy and/or redundancy.

A block diagram of the hardware components on each quadcopter and their connections can be seen in Figure 2. There is one custom main controller board that is responsible for the high-speed (1000 Hz) attitude control loop. The position control loop and part of the position estimation is also done here. The components and their functions on the mainboard are:

- The STM32F4 microcontroller is responsible for all computation and communicates with the other components on the mainboard.
- The MPU9150 IMU sensor provides the raw data that is used for attitude estimation. It has a three-axis accelerometer, a three-axis gyroscope, and a three-axis magnetometer; thereby providing nine degrees of freedom.
- The barometer measures the air pressure and is currently not used in any algorithm. Later, it could be used for redundancy when measuring the altitude.
- The CC2520 radio transceiver is used to communicate with the ground station and other quadcopters.
- Standard Pulse-Position Modulation (PPM) signals are sent to motor controllers that drive the propeller motors.

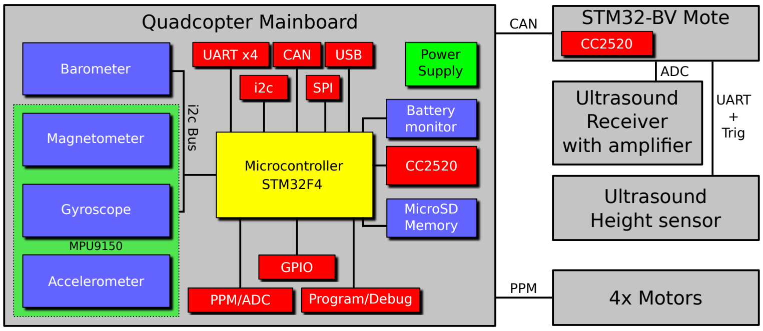


Figure 2: The hardware setup on each quadcopter.

The STM32BV-mote is responsible for clock synchronization and distance measuring. An ultrasound receiver is connected to an Analog to Digital Converter (ADC) pin with a simple amplifier to capture pulses from the anchors. The mote also communicates with an ultrasound altitude sensor to measure the distance to the floor. These measurements together are sent to the mainboard that computes the position of the copter based on them.

4 Ultrasound Distance Measurement

The ultrasound distance measurements are done by synchronizing the clocks of all anchors and quadcopters and having timeslots assigned when pulses are sent from different anchors, which are then recorded by the receivers on the quadcopters. The receiver then uses the ToF of the pulse to calculate the distance to the anchor.

Clock synchronization is done by having a node sending out a clock value and using a hardware interrupt on the receiving nodes that saves the local clock value at the time the packet starts being received. When the whole clock packet is received, the difference between the time stamp and the received clock value is subtracted from the own clock. This difference is also used to estimate the clock drift and compensate for that over time. With clock packets sent every 2 s, the clock has a jitter of less than 5 µs, which is good enough to measure the ToF of sound.

In order to reject noise on the ultrasound measurements, the pulses sent out by the anchors are created by multiplying the 40 kHz carrier with a sinc pulse. The received signal is then cross-correlated with the same sinc pulse to find the first peak above a certain threshold. In order to speed up the cross correlation, it is performed using overlapping Fast Fourier Transforms (FFTs) [18]. Figure 3 shows the recorded ultrasound pulse and the cross correlation result from an anchor that is placed 10 m away. It can be seen that the noise amplitude is rejected on the correlated signal, making analysis of the distance easier.

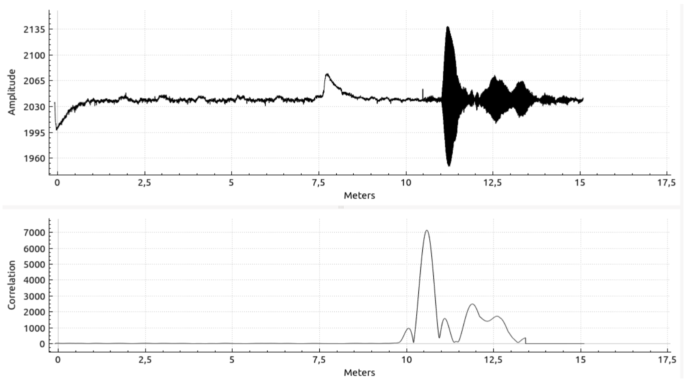


Figure 3: Ultrasound samples recorded for a time corresponding to 10 meters and the cross correlation result.

5 Position Estimation

The software on the quadcopter mainboard does the bulk of all the computational work required for the copter to operate. This section gives a brief overview of the discrete-time calculation performed in software to update the state of the system at time n regularly with interval dt.

The algorithm that runs at the highest rate of the control system is the attitude estimation and control. We have used a slightly modified version of an Attitude and Heading Reference System (AHRS) algorithm [19] to get a quaternion-based representation of the current attitude, from which we calculate Euler angles as:

$$\begin{bmatrix} \theta_r(n) \\ \theta_p(n) \\ \theta_y(n) \end{bmatrix} = \begin{bmatrix} atan2(2(q_0q_1 + q_2q_3), 1 - 2(q_1^2 + q_2^2)) \\ arcsin(2(q_0q_2 - q_3q_1)) \\ atan2(2(q_0q_3 + q_1q_2), 1 - 2(q_2^2 + q_3^2)) \end{bmatrix} \tag{1}$$

where $[q_0, q_1, q_2, q_3]^T$ is the quaternion representation of the current attitude, $atan2$ is a function for $arctan$ that takes two arguments to handle all possible angles and $[\theta_r(n), \theta_p(n), \theta_y(n)]^T$ are the roll, pitch, and yaw Euler angles. Then, there is one Proportional-Integral-Derivative (PID) controller for each Euler angle to stabilize the copter. There is also a PID controller for the altitude. In order to get as little altitude-variations as possible, feed-forward is used on the throttle output from the roll and pitch-angles, calculated as:

$$FF_{fac}(n) = \sqrt{tan(\theta_r(n))^2 + tan(\theta_p(n))^2 + 1} \tag{2}$$

The feed-forward term $FF_{fac}(n)$ is calculated at a higher rate than the altitude measurements arrive and represents a compensation factor that makes the vertical thrust component constant while the roll and pitch angles $[\theta_r(n), \theta_p(n)]^T$ vary. This equation has singularities when the roll or pitch angles are at 90°, but the attitude control loop truncates its inputs to prevent the roll and pitch angles from exceeding 45°.

After several manual aggressive flight tests with altitude hold activated, we had confidence that our altitude control loop was working properly.

To estimate the position of the copter, we use one high-rate update based on dead reckoning from its attitude. The assumption is that the throttle is controlled such that the altitude remains constant or slowly changing. Additionally, one low-rate update is used on the position every time new ultrasound ranging values arrive. For the high-rate dead reckoning, the first thing we calculate for each iteration is the velocity-difference $[d_{vx}(n), d_{vy}(n)]^T$ and add it to the integrated velocity value $[V_x(n), V_y(n)]^T$, rotated by the yaw angle:

$$\begin{bmatrix} d_{vx}(n) \\ d_{vy}(n) \end{bmatrix} = \begin{bmatrix} 9.82tan(\theta_r(n) + \theta_{rofs}(n))dt \\ 9.82tan(\theta_p(n) + \theta_{pofs}(n))dt \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} c_y \\ s_y \end{bmatrix} = \begin{bmatrix} cos(\theta_y(n)) \\ sin(\theta_y(n)) \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} V_x(n) \\ V_y(n) \end{bmatrix} = \begin{bmatrix} V_x(n-1) + d_{vx}(n)c_y + d_{vy}(n)s_y \\ V_y(n-1) + d_{vx}(n)s_y + d_{vy}(n)c_y \end{bmatrix} \tag{5}$$

where $\theta_{rofs}(n)$ and $\theta_{pofs}(n)$ are offsets that could be estimated over time to compensate for misalignment of the accelerometer. Again, the singularity when the roll or pitch angle $[\theta_r(n), \theta_p(n)]^T$ are 90° is not an issue because these angles are limited at 45°. This is then used to update the position $[P_x(n), P_y(n)]^T$:

$$\begin{bmatrix} P_x(n) \\ P_y(n) \end{bmatrix} = \begin{bmatrix} P_x(n-1) + V_x dt \\ P_y(n-1) + V_y dt \end{bmatrix} \tag{6}$$

As the velocity integration drift is unbounded even when there is a small offset on the attitude estimation, the anchor distance measurements have to be used to estimate the velocity drift in addition to the roll and pitch error. For the anchor corrections, which arrive at a lower rate, we first compute the difference between the expected distance to the anchor from the dead-reckoning and the measured distance to the anchor:

$$\begin{bmatrix} d_{ax}(n) \\ d_{ay}(n) \\ d_{az}(n) \end{bmatrix} = \begin{bmatrix} P_x(n-1) - P_{x,anchor} \\ P_y(n-1) - P_{y,anchor} \\ P_z(n-1) - P_{z,anchor} \end{bmatrix} \tag{7}$$

$$d_a(n) = \sqrt{d_{ax}(n)^2 + d_{ay}(n)^2 + d_{az}(n)^2} \tag{8}$$

$$err(n) = d_a(n) - d_{measured}(n) \tag{9}$$

$$F_c(n) = \frac{err(n)}{d_a(n)} \tag{10}$$

where $[P_x(n), P_y(n), P_z(n)]^T$ is the position of the copter, $[P_{x,anchor}, P_{y,anchor}, P_{z,anchor}]^T$ is the position of the anchor this measurement came from and $[d_{ax}(n), d_{ay}(n), d_{az}(n)]^T$ is the difference between them. Further, $d_a(n)$ is the magnitude of the calculated difference, $d_{measured}(n)$ is the measured magnitude, $err(n)$ is the difference between the calculated and the measured magnitude and $F_c(n)$ is a factor that is used in later calculations for correction. Notice that there is a singularity when $d_a(n)$ approaches 0, but this would imply that the copter is located exactly on one anchor which means that the copter collides with that anchor. This should not happen because the copters should keep a safety distance from the anchors at all times.

At this point, if the error is larger than a certain threshold, we discard this measurement and lower the position quality because something is likely to be wrong. If too many consecutive measurements have a large error, we stop discarding and start using them in case this is the initial position correction at start-up.

Next, the position differences $[d_{ax}(n), d_{ay}(n), d_{az}(n)]^T$ are used to correct the current position and the velocity error where we compute proportional and derivative parts, $[P_{xpos}(n), P_{ypos}(n)]^T$ and $[D_{xpos}(n), D_{ypos}(n)]^T$, on the position error. The gain components in the following equations $(G_{p,vel}, G_{p,pos}, G_{d,pos})$ were derived experimentally and the simulation presented in Section 7 has been an important aid for doing that.

$$\begin{bmatrix} P_{xpos}(n) \\ P_{ypos}(n) \end{bmatrix} = \begin{bmatrix} d_{ax}(n)F_c G_{p,pos} \\ d_{ay}(n)F_c G_{p,pos} \end{bmatrix} \tag{11}$$

$$\begin{bmatrix} D_{xpos}(n) \\ D_{ypos}(n) \end{bmatrix} = \begin{bmatrix} (d_{ax}(n)F_c - d_{ax}(n-1)F_c)G_{d,pos} \\ (d_{ay}(n)F_c - d_{ay}(n-1)F_c)G_{d,pos} \end{bmatrix} \tag{12}$$

Then, apply this to the position:

$$\begin{bmatrix} P_x(n) \\ P_y(n) \end{bmatrix} = \begin{bmatrix} P_x(n-1) + P_{xpos}(n) + D_{xpos}(n) \\ P_y(n-1) + P_{ypos}(n) + D_{ypos}(n) \end{bmatrix} \tag{13}$$

The height $P_z(n)$ could also be updated as above, but using the ultrasound sensor directed towards the floor directly gave better results in our experiments.

Updating the velocity state $[V_x(n), V_y(n)]^T$ is done in a similar way:

$$\begin{bmatrix} V_x(n) \\ V_y(n) \end{bmatrix} = \begin{bmatrix} V_x(n-1) + d_{ax}F_c(n)G_{p,vel} \\ V_y(n-1) + d_{ay}F_c(n)G_{p,vel} \end{bmatrix} \tag{14}$$

A test flight of 60 s where a simple PID control loop is issuing control commands to hold the $[x, y]^T$ position based on the estimated position is shown in Figure 4. The overlapping red dots represent estimated position samples during this flight, and it can be seen that the deviation was below 20 cm for the entire flight. Notice that we did not have a more accurate positioning system to compare with in this test. The distribution of the estimated position gives an impression about the performance.

Because of the complexity we did not attempt to make an analytical stability analysis of the position-estimation algorithm. We did an experimental stability analysis using fault injection presented in Section 7.1.

6 Collision Avoidance

In this study, collision avoidance is attempted by placing risk contours around copters and static objects from the perspective of every copter, and steering away if the risk contours overlap with the comfort zone of the copter [20]. This means that the risk contours are not a global state, but different from every copter's perspective based on its relative velocity to the object and when the

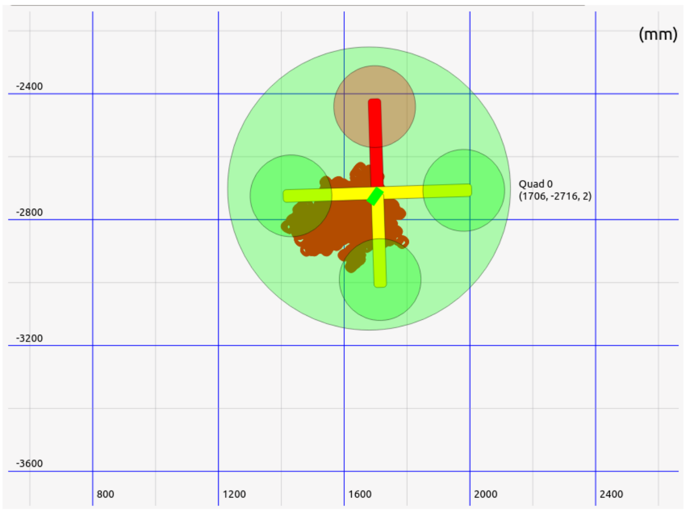


Figure 4: The estimated position during a 60 s long hover.

positions of other copters were last received. The comfort zone is represented as a circle placed around the quadcopter with a radius that is calculated based on the confidence of the position estimation.

The risk contours are represented as ellipses and sized/rotated based on the squared relative velocity vector to the copters/objects they surround. To share the knowledge about the position of all copters, they broadcast this information one at a time to everyone else. When a copter receives a position update from another copter, it will update its Local Dynamic Map (LDM) with this information. Between the position updates, the risk contours around other copters will be moved and reshaped based on the velocity the other copters had when their position was last received. What this looks like can be seen in the screenshot in Figure 5 of the visualization and control program we developed for this application.

The risk contour around every neighbouring object in each copter's LDM looks like the following:

$$\begin{bmatrix} d_{vx} \\ d_{vy} \end{bmatrix} = \begin{bmatrix} V_{x,r} - V_x \\ V_{y,r} - V_y \end{bmatrix} \tag{15}$$

$$d_v = \sqrt{d_{vx}^2 + d_{vy}^2} \tag{16}$$

where $[d_{vx}, d_{vy}]^T$ are the X and Y velocity difference between this copter's own velocity and the velocity $[V_{x,r}, V_{y,r}]^T$ of the copter corresponding to this risk contour in the LDM. The position $[R_{px}, R_{py}]^T$, width, height $[R_w, R_h]^T$ and rotation θ_r of the risk contour are calculated as:

$$\begin{bmatrix} R_{px} \\ R_{py} \end{bmatrix} = \begin{bmatrix} P_{x,c} + R_{gx} d_{vx} d_v \\ P_{y,c} + R_{gx} d_{vy} d_v \end{bmatrix} \tag{17}$$

$$\begin{bmatrix} R_w \\ R_h \end{bmatrix} = \begin{bmatrix} R_r + R_{gx} d_{vx} d_v^2 \\ R_r + R_{gy} d_{vy} d_v^2 \end{bmatrix} \tag{18}$$

$$\theta_r = atan2(d_{vy}, d_{vx}) \tag{19}$$

where R_r is a safety margin around the copter in the LDM that this risk contour surrounds. R_r is scaled based on the time that has passed since the copter corresponding to this risk contour was heard the last time. $[P_{x,c}, P_{y,c}]^T$ is the position of the copter corresponding to this risk contour. Further, R_{gx} and R_{gy} are factors that scale the size of the risk contour that we found suitable values for in the auto-generated tests described in Section 7.

When an overlap between the comfort zone of a copter and a risk contour occurs, the collision-avoidance mechanism will take over control and steer away from the overlapping risk contour in the opposing direction. If there are several simultaneous overlaps, a vector will be calculated from a weighted sum of all overlapping risk contours and their relative direction, and used to steer away from the collision, calculated as:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \sum_{i=0}^{N} \begin{bmatrix} C_{x,i} M_i \\ C_{y,i} M_i \end{bmatrix} \tag{20}$$

$$\begin{bmatrix} C_r \\ C_p \end{bmatrix} = \begin{bmatrix} -cos(\theta_y) C_x - sin(\theta_y) C_y \\ -sin(\theta_y) C_x + cos(\theta_y) C_y \end{bmatrix} \tag{21}$$

where $[C_x, C_y]^T$ are the relative $[X, Y]^T$ direction sums of all risk contours i that overlap with the comfort zone of the copter. $[C_r, C_p]^T$ are the roll and pitch output commands calculated from all overlapping risk contours, rotated by the yaw angle θ_y of the copter. Further, M_i is the amount of overlap with every overlapping risk contour i. Thus, the more overlap there is for one risk contour, the more influence it will have on the output.

It should be noted that collision avoidance is done in two dimensions. This is because our copters are not able to fly over each other even if they are at different heights, since the height sensor of each copter requires a free path to the ground. Since the position-estimation algorithm relies on an altitude controller that keeps the altitude constant or slowly changing, collision avoidance in the Z direction is not necessary if truncation is used on the set point of the altitude controller.

7 Simulation and Fault Injection

To evaluate and optimize our quadcopter system, we have created a simulator with the architecture shown in Figure 6. Our simulator is a library written in C++ with an interface where copters can be added, removed, or commanded to move. The block named *coptermodel* runs the same code for position and velocity estimation, shown in Equation 5 and 6, as the real implementation on the hardware copters. The angles $[\theta_r, \theta_p, \theta_y]^T$ are updated from the movement command with a similar response to that of the actual hardware, and then the position and velocity state is updated based on these angles. For this update,

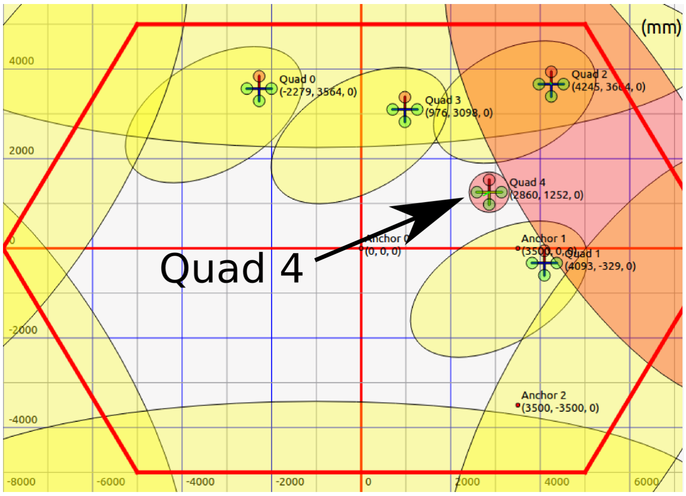


Figure 5: A screenshot of the risk contours from the perspective of Quad 4. The red contour is red because there is an overlap between the copters own comfort zone (Quad 4) and the risk contour around the right upper wall.

we do not inject any faults and assume that it represents the true position of the copter.

There is also a position and velocity state that is updated in the same way, but where we inject various faults. This perceived position is then corrected from simulated ultrasound measurements as described in Equation 13 and 14, while we inject faults on these ultrasound measurements. Additionally, each simulated copter has the collision-avoidance mechanism described in Section 6, shown as Intelligent Transportation System (ITS)-station in Figure 6. The simulated ITS-station on each copter broadcasts and receives ITS-messages to and from the other copters every 100 ms, where we also inject faults. The CopterSim library can either be used from a Graphical User Interface (GUI) to manually add and move copters, or from a program that auto-generates tests and injects faults. All fault injection is done with probes from the FaultCheck tool [10], linked to the simulator.

We have created a model for the QuickCheck tool [11] that sends commands to the simulator where we add a random number of copters at random non-overlapping positions and run commands while checking the property that they do not collide. These randomly-generated commands can either be steering commands for the copters, or fault-injection commands passed to the FaultCheck tool. The whole set-up can be seen in Figure 7.

The parameters for the steering commands are:

- Which copter to command, randomly chosen from all the copters present in the simulation.
- The roll output, chosen between $\pm 15°$.

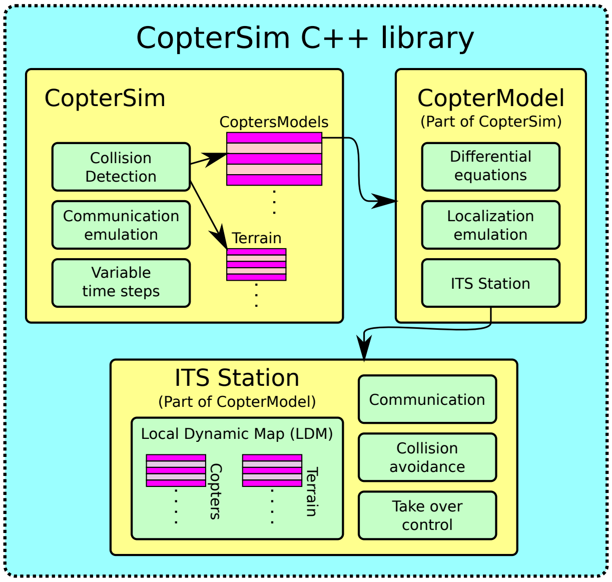


Figure 6: The simulation architecture. CopterSim has a list with all copters and checks for collisions between them and the terrain in every iteration. CopterModel handles the physical model of every copter and has an ITS-station that handles collision avoidance. The localization emulation is also part of CopterModel.

- The pitch output, chosen between ±15°.
- The yaw rate output, randomly chosen between ±90° per second.

Further, the fault injection commands have the following parameters:

- Which copter to affect, randomly chosen from all the copters present in the simulation.
- The fault type, randomly chosen from:
  - Communication bit-flip, which flips a randomly chosen bit of the broadcast ITS message.
  - Packet loss, where ITS-messages are lost.
  - Repetition, where ITS-messages are repeated.
  - Ultrasound ranging faults, where a random offset is added to the ultrasound distance measurements.
  - Offsets on the $[\theta_r, \theta_p, \theta_y]^T$ angles.

When a collision occurred during these auto-generated tests, we used the QuickCheck tool to shrink the sequence of commands to a smaller one that led to a collision, in order to make it easier to identify the problem. Then, we replayed this smaller sequence of commands while adjusting the gains described

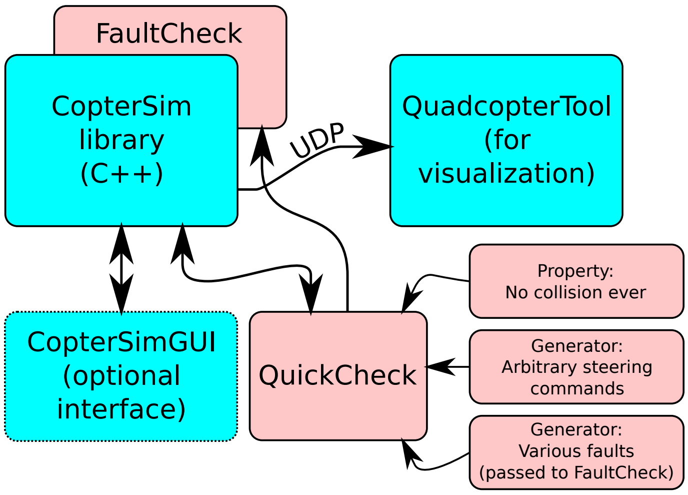

FaultCheck
CopterSim library (C++)
UDP
QuadcopterTool (for visualization)
Property: No collision ever
CopterSimGUI (optional interface)
QuickCheck
Generator: Arbitrary steering commands
Generator: Various faults (passed to FaultCheck)

Figure 7: CopterSim with FaultCheck connected to a visualization tool and QuickCheck.

in Section 5 and the risk contour parameters described in Section 6 until this command sequence did not lead to a collision any more. This also gave us insight into the number of simultaneous faults the copters can handle.

7.1 Experimental Performance and Stability Analysis

Here we show specific injected faults and their impact on the position and velocity error under different gain values, which are described in Section 5. This is not a full analysis of all possible combinations of faults and gains, but it gives a general impression about the fault tolerance and performance of the system under different conditions. Running many auto-generated tests with different combinations of injected faults gave us confidence that the chosen parameters gave a robust position correction.

Figure 8 shows how the position recovers when a position offset fault of 1.5 m is injected. The left part of the graph shows the recovery with $G_{p,pos} = 0.2$ and the right part with $G_{p,pos} = 1.0$. It can be seen that the higher position gain makes the position error recover faster. A similar relation is shown in Figure 9 where a pitch offset is injected for different values of $G_{p,vel}$. When the gain is too high, an oscillation such as in Figure 10 where a position offset of 1.5 m is injected while $G_{p,pos} = 2.0$ can occur.

In Figure 11 a position offset fault is shown with $G_{p,vel} = 0.0$ and $G_{p,vel} = 2.0$. It can be seen that when only a position offset is injected, the velocity gain does not help at all. However, a roll or pitch offset such as in Figure 9 requires $G_{p,vel} > 0$ to recover. An example where both a ranging offset and a pitch offset are injected at the same time can be seen in Figure 12, where the difference between low and high $G_{p,vel}$ can be seen. The injected pitch fault requires $G_{p,vel} > 0$, but the ranging fault recovers the same way with lower $G_{p,vel}$.

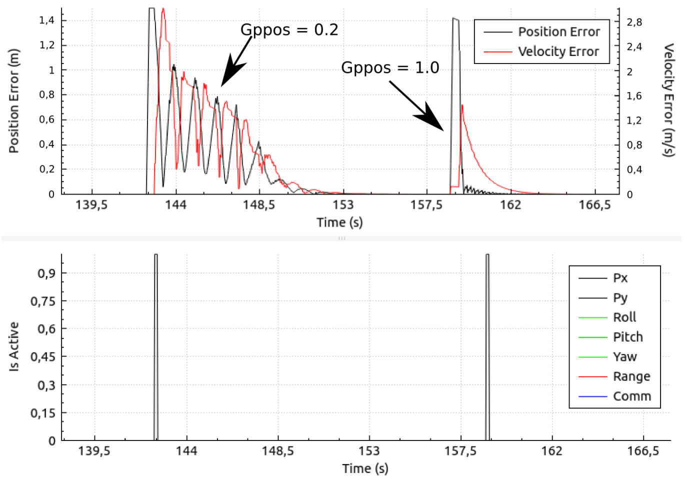


Figure 8: Fault injection with 1.5 m position offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,pos} = 0.2$ and the second one had $G_{p,pos} = 1.0$.

A dynamic example, where one copter is moved forth and back, is shown in Figure 13 for different values of $G_{p,pos}$ while an amplification on the pitch of 0.95 is injected. If there were not any acceleration the pitch amplification fault would remain unnoticed, but the acceleration makes it appear. It can be seen that higher gain keeps the position error lower during the flight.

8 Conclusions

We have created a quadcopter platform that has a novel approach to localization using ultrasound distance measurement combined with IMU-based dead reckoning for accurate positioning, while we use risk contours to avoid collisions with static objects and other copters. Additionally, we have created a powerful simulation environment where we can auto-generate tests and inject faults with many copters simultaneously, making it possible to scale up the tests beyond what our hardware allows. Our current platform has a limited size, because the anchors can be no further away from the copters than 12 m. Future work includes implementation of handover, both in simulation and hardware, between flying zones to handle more anchors spread out in a larger area. Another improvement would be handover between GNSS positioning and the positioning method proposed in this work, when higher position accuracy is required during landing and take-off.

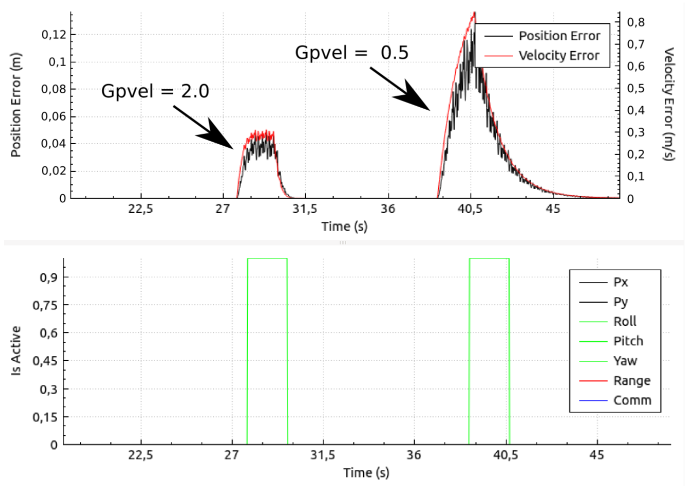


Figure 9: Fault injection with 5° pitch offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 2.0$ and the second one had $G_{p,vel} = 0.5$.

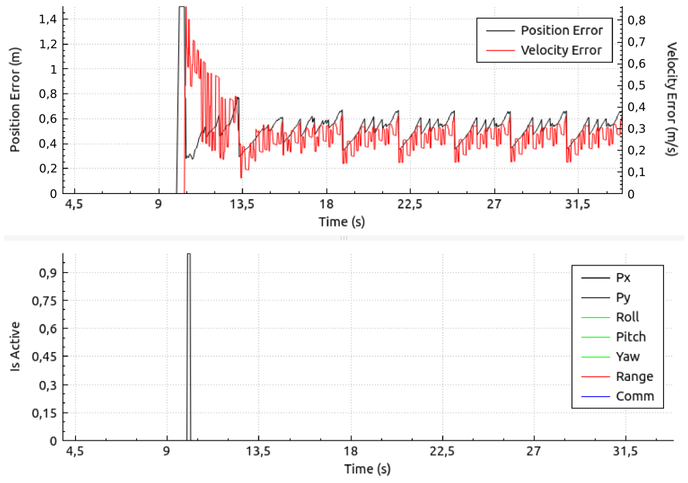


Figure 10: Fault injection with 1.5 m position offset and $G_{p,pos} = 2.0$. The lower part shows when the faults are active and the upper part shows the position and velocity errors.

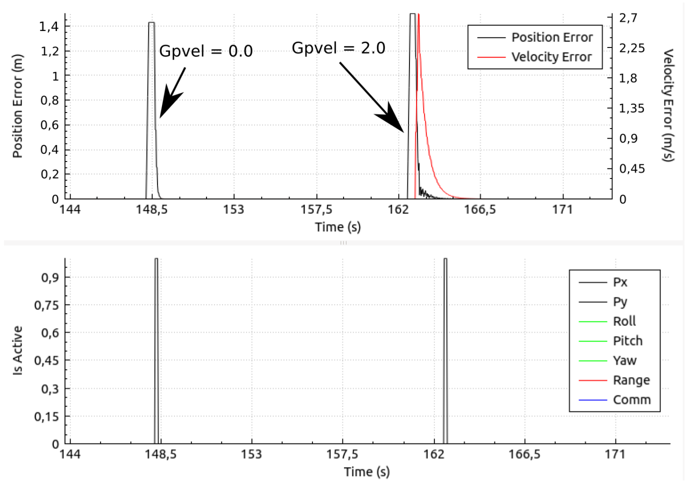


Figure 11: Fault injection with 1.5 m position offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 0.0$ and the second one had $G_{p,vel} = 2.0$.

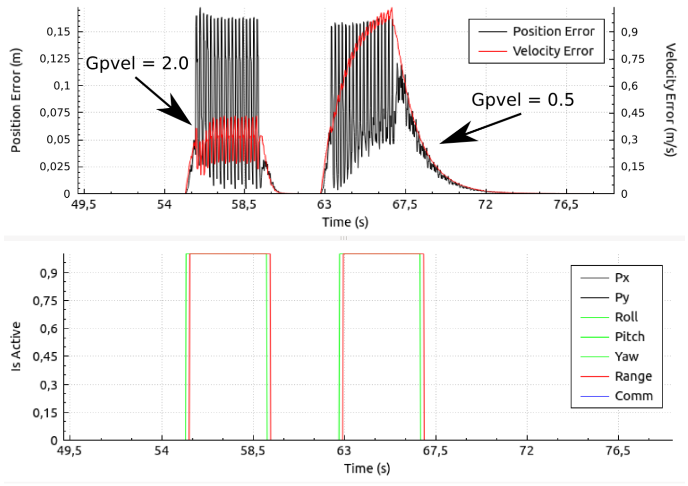


Figure 12: Fault injection with 5° pitch offset and 0.5 m anchor distance offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 2.0$ and the second one had $G_{p,vel} = 0.3$.

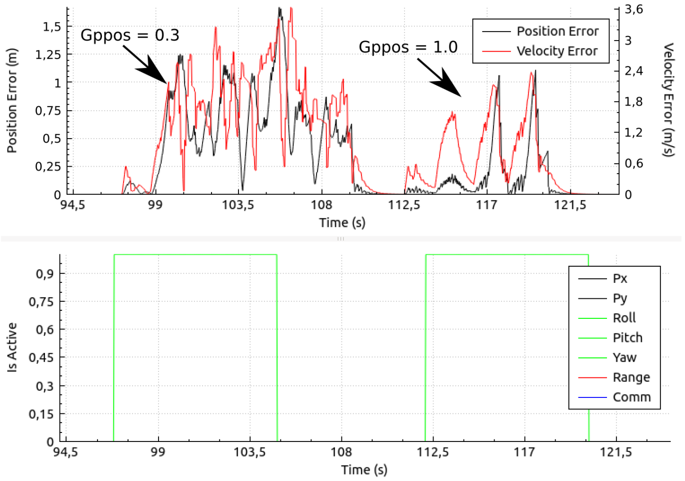


Figure 13: Fault injection with 0.95 pitch amplification. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,pos} = 0.3$ and the second one had $G_{p,pos} = 1.0$.

9 Acknowledgement

This research has been funded through the KARYON EU project (Grant agreement no: 288195), the PROWESS EU project (Grant agreement no: 317820) and through EISIGS (grants from the Knowledge Foundation). We would like to thank Kenneth Östberg for fruitful discussions regarding the usage of risk contours.

References

[1] M. Achtelik, T. Zhang, K. Kuhnlenz, and M. Buss. ”Visual Tracking and Control of a Quadcopter Using a Stereo Camera System and Inertial Sensors”. In: *Proceedings of the International Conference on Mechatronics and Automation (ICMA)*. 2009, pp. 2863–2869.

[2] B. Ben Moshe, N. Shvalb, J. Baadani, I. Nagar, and H. Levy. ”Indoor Positioning and Navigation for Micro UAV Drones”. In: *Proceedings of the 27th Convention of Electrical Electronics Engineers in Israel (IEEEI)*. 2012, pp. 1–5.

[3] M. Bošnak, D. Matko, and S. Blažič. ”Quadrocopter Hovering Using Position-Estimation Information from Inertial Sensors and a High-delay Video System”. In: *Journal of Intelligent & Robotic Systems* 67.1 (2012), pp. 43–60.

[4] J. Engel, J. Sturm, and D. Cremers. ”Accurate Figure Flying with a Quadrocopter Using Onboard Visual and Inertial Sensing”. In: *Proc. of the Workshop on Visual Control of Mobile Robots (ViCoMoR) at the IEEE/RJS International Conference on Intelligent Robot Systems (IROS)*. 2012.

[5] S. Grzonka, G. Grisetti, and W. Burgard. ”Towards a Navigation System for Autonomous Indoor Flying”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).* 2009, pp. 2878–2883.

[6] I. Sa and P. Corke. ”System Identification, Estimation and Control for a Cost Effective Open-Source Quadcopter”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).* 2012, pp. 2202–2209.

[7] B. Vedder. *CC2520 and STM32F4 RF Boards.* 2014. URL: http://vedder.se/2013/04/cc2520-and-stm32-rf-boards/.

[8] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell. ”Fault Injection for Dependability Validation: A Methodology and Some Applications”. In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 166–182.

[9] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects.* Ed. by F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 250–271.

[10] 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.* ES4CPS ’14. Dresden, Germany: ACM, 2014, 1:1–1:8.

[11] T. Arts, J. Hughes, J. Johansson, and U. Wiger. ”Testing Telecoms Software with Quviq QuickCheck”. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang.* Portland, Oregon: ACM Press, 2006.

[12] G. Hoffmann, D. Rajnarayan, S. Waslander, D. Dostal, J. S. Jang, and C. Tomlin. ”The Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control (STARMAC)”. In: *The 23rd Digital Avionics Systems Conference, DASC 04.* Vol. 2. 2004, pp. 4–121.

[13] J. Pestana, J. Sanchez-Lopez, P. de la Puente, A. Carrio, and P. Campoy. ”A Vision-Based Quadrotor Swarm for the Participation in the 2013 International Micro Air Vehicle Competition”. In: *Proceedings of the International Conference on Unmanned Aircraft Systems (ICUAS).* 2014, pp. 617–622.

[14] J. Eckert, R. German, and F. Dressler. ”On Autonomous Indoor Flights: High-Quality Real-Time Localization Using Low-Cost”. In: *IEEE International Conference on Communications (ICC 2012), IEEE Workshop on Wireless Sensor Actor and Actuator Networks (WiSAAN 2012).* Ottawa, Canada: IEEE, 2012, pp. 7093–7098.

[15] J. F. Roberts, T. Stirling, J. Zufferey, and D. Floreano. ”Quadrotor Using Minimal Sensing for Autonomous Indoor Flight”. In: *European Micro Air Vehicle Conference and Flight Competition (EMAV2007).* Toulouse, France, 2007.

[16] J. Eckert, R. German, and F. Dressler. ”ALF: An Autonomous Localization Framework for Self-Localization in Indoor Environments”. In: *7th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2011).* Barcelona, Spain: IEEE, 2011, pp. 1–8.

[17] J. Eckert. *Autonomous Localization Framework for Sensor and Actor Networks: Autonomes Lokalisierungsframework Für Sensor- und Aktornetzwerke.* 2012.

[18] J. Jan. *Digital Signal Filtering, Analysis and Restoration*. IEE telecommunications series. Institution of Electrical Engineers, 2000.

[19] S. Madgwick, A. J. L. Harrison, and R. Vaidyanathan. ”Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm”. In: *IEEE International Conference on Rehabilitation Robotics (ICORR)*. 2011, pp. 1–7.

[20] K. Ö. et al. *Safety Constraints and Safety Predicates*. Public Report. KARYON consortium, 2014.

Paper III

Using Simulation, Fault Injection and Property-Based Testing to Evaluate Collision Avoidance of a Quadcopter System

Benjamin Vedder, Jonny Vinter and Magnus Jonsson

Published in Proceedings of the 1st Workshop on Safety and Security of Intelligent Vehicles (SSIV), Rio de Janeiro, Brazil, 2015

Abstract

In this work we use our testing platform based on FaultCheck and QuickCheck that we apply on a quadcopter simulator. We have used a hardware platform as the basis for the simulator and for deriving realistic fault models for our simulations. The quadcopters have a collision-avoidance mechanism that shall take over control when the situation becomes hazardous, steer away from the potential danger and then give control back to the pilot, thereby preventing collisions regardless of what the pilot does. We use our testing platform to randomly generate thousands of simulations with different input stimuli (using QuickCheck) for hundreds of quadcopters, while injecting faults simultaneously (using FaultCheck). This way, we can effectively adjust system parameters and enhance the collision-avoidance mechanism.

1 Introduction

For safety-critical systems, non-functional requirements such as fault tolerance have to be considered. One way to evaluate and exercise fault tolerance mechanisms is by using Fault Injection (FI) [1]. FI can e.g. be carried out early in the development process for models of hardware [2–4], models of software [5–8], source code [9, 10], and at later stages of the development process, for software deployed on target hardware [11–16]. When working with FI, it is common to manually create input stimuli for a System Under Test (SUT) and run it without faults, and save the state of the SUT during this run as the *golden run*. After that, the experiment is repeated with the same input stimuli again while injecting faults, and the system state is compared to the corresponding state from the golden run. This will show how faults affect the SUT for a pre-defined input sequence. In our work, we automatically generate the input stimuli to find out how the system behaves when faults are injected under different conditions.

When dealing with software testing, one way to make sure that functional requirements are fulfilled is using Property-Based Testing (PBT) [17]. When doing PBT, inputs are automatically generated and a model of the software is used to evaluate whether it fulfils its specification, whereby the golden run is generated automatically for each test sequence based on the model. Previously, we have introduced the concept of combining techniques from the areas of PBT and FI using the commercially available PBT-tool QuickCheck [18] and our FI-tool FaultCheck to test functional and non-functional requirements simultaneously [10]. By using techniques from PBT while doing FI, we can automatically generate golden runs during our experiments and test the SUT using thousands of input sequences and fault combinations. The aim of this work is to evaluate how effectively our testing platform, based on FaultCheck and QuickCheck, can be used during the development of a complex SUT while doing FI with realistic fault models.

The SUT that we are using is a quadcopter simulator that is based on the hardware quadcopter platform described in Section 2. We have derived several realistic fault models from the hardware platform that we inject during the simulation to make sure that the real quadcopters can deal with these faults without collisions. The simulated quadcopters have a collision-avoidance mechanism that automatically takes over control if the situation becomes dangerous in order to avoid a collision with the terrain or other quadcopters. As soon as the collision is avoided, control is be given back to the pilot, who can be a human or an autonomous system. This collision-avoidance system relies on communication between the quadcopters and knowledge of each quadcopters current position. To test the functional requirements of this system, we randomly place quadcopters in an environment and give them random steering commands using QuickCheck. The postcondition for each generated test to succeed is that the copters should never collide regardless of their steering commands. When these auto-generated simulations work as expected, we run them again while injecting faults using FaultCheck. This helps us to figure out problematic scenarios when certain faults are present and to add fault handling mechanisms and safety margins so that the system can deal with faults during these scenarios.

When simulating the quadcopters, a physical model with differential equations is used to calculate their positions and movements. This means that their positions are always known, which makes it difficult to test the position-estimation algorithm. One way to test their position estimation is to start the simulator with a "true" (golden run) position state and a perceived (incorrect) position state, while simulating sensor readings from their correct position to evaluate how their perceived position converges to the correct position. Creating the perceived position can obviously be done manually, but FaultCheck provides a variety of different fault models to chose from in order to create a faulty position from the correct position. This way, our testing platform allows us to test the collision-avoidance mechanism and the position-estimation algorithm simultaneously. This provides a more realistic scenario than testing the individual parts of the system isolated from each other.

Compared to the normal case with QuickCheck, where the golden run is calculated in the model, our simulator calculates the true position continuously while the simulations are running. Calculating the true position within the simulator gives advantages because the calculation framework is already present in the simulator and does not have to be reimplemented in the QuickCheck model again. Another advantage is that this gives improvements in execution speed in our case since the simulator is written in the language C++, which is designed for high performance. Since the differential equations of all simulated copters are evaluated hundreds of times every simulated second, having high execution speed is important to run long simulations, with many copters, in a reasonable amount of time.

The contributions of this work are as follows:

- We show how to derive realistic fault models based on the hardware quadcopter platform that the simulator is based on, and how to relate them to the equations of the movement and position updates of the quadcopters. We also show how to represent and inject these faults into the simulator using FaultCheck.
- We show a method to intuitively visualize failed test cases that lead to a collision between the quadcopters. Since the visualization is created

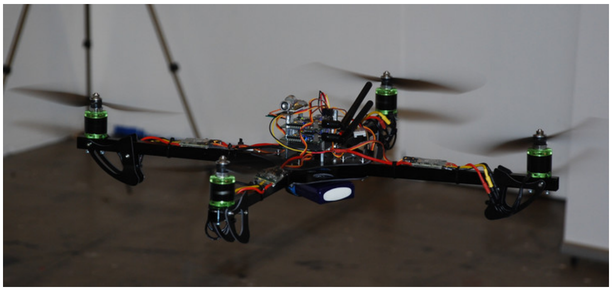


Figure 1: A photo of one of the quadcopters.

in real-time based on a list of QuickCheck commands (e.g. steering and FI) that lead to the collision, we can adjust and enhance the collision-avoidance mechanism and replay and visualize the experiment with the same commands over and over again attempting to avoid a collision caused by this series of commands.

- We show how our testing platform based on FaultCheck together with QuickCheck scales when testing a complex SUT, namely the quadcopter simulator.

The rest of this paper is organized as follows. In Section 2 we describe the quadcopter system that our simulator is based on. In Section 3 we describe our simulator and in Section 4 we show how we apply our testing platform on the quadcopter simulator. Further, in Section 5 we show how we visualize and deal with failed test sequences, and in Section 6 we present our conclusions from this work.

2 Quadcopter System

The hardware quadcopter platform that our simulator is based on consists of four quadcopters, as the copter shown in Figure 1, and two or more anchors that are placed at known locations. The anchors send ultrasound pulses to the copters at certain timeslots and are clock synchronized with the copters. Based on the time when the ultrasound pulses are received by the copters, they calculate the time of flight of the pulses and hence the distance to the anchors. A drawing of the hardware quadcopter platform and the anchors can be seen in Figure 2.

Each quadcopter has two computing nodes connected over a CAN bus. The main computing node handles 1) attitude estimation and control, 2) position estimation and 3) collision avoidance. The second computing node is responsible for clock synchronization and measuring the distance from the copter to the floor and the distance to the anchors.

To give an insight about the connection between the quadcopter system and the simulator, we describe the discrete-time equations that the quadcopter uses to estimate its position. Every time n with interval dt the inertial sensors on the quadcopter are sampled to update the state of the quadcopter.

The algorithm that runs at the highest rate of the control system is the attitude estimation and control. We have used a slightly modified version of an Attitude and Heading Reference System (AHRS) algorithm [19] to get a quaternion-based representation of the current attitude, from which we calculate Euler angles as:

$$\begin{bmatrix}\theta_r(n)\\ \theta_p(n)\\ \theta_y(n)\end{bmatrix} = \begin{bmatrix} atan2(2(q_0q_1+q_2q_3), 1-2(q_1^2+q_2^2)) \\ arcsin(2(q_0q_2-q_3q_1)) \\ atan2(2(q_0q_3+q_1q_2), 1-2(q_2^2+q_3^2)) \end{bmatrix} \tag{1}$$

where $[q_0, q_1, q_2, q_3]^T$ is the quaternion representation of the current attitude, $atan2$ is a function for $arctan$ that takes two arguments to handle all possible angles and $[\theta_r(n), \theta_p(n), \theta_y(n)]^T$ are the roll, pitch, and yaw Euler angles. Then, there is one Proportional-Integral-Derivative (PID) controller for each Euler angle to stabilize the copter. There is also a PID controller for the altitude. In order to get as little altitude-variations as possible, feed-forward is used on the throttle output from the roll and pitch-angles, calculated as:

$$FF_{fac}(n) = \sqrt{tan(\theta_r(n))^2 + tan(\theta_p(n))^2 + 1} \tag{2}$$

The feed-forward term $FF_{fac}(n)$ is calculated at a higher rate than the altitude measurements arrive and represents a compensation factor that makes the vertical thrust component constant while the roll and pitch angles $[\theta_r(n), \theta_p(n)]^T$ vary.

To estimate the position of the copter, we use one high-rate update based on dead reckoning from its attitude. The assumption is that the throttle is controlled such that the altitude remains constant or slowly changing. Additionally, one low-rate update is used on the position every time new ultrasound ranging values arrive from the anchors.

For the high-rate dead reckoning, the first thing we calculate for each iteration is the velocity-difference $[d_{vx}(n), d_{vy}(n)]^T$ and add it to the integrated velocity value $[V_x(n), V_y(n)]^T$, rotated by the yaw angle:

$$\begin{bmatrix}d_{vx}(n)\\ d_{vy}(n)\end{bmatrix} = \begin{bmatrix}9.82tan(\theta_r(n)+\theta_{rofs}(n))dt\\ 9.82tan(\theta_p(n)+\theta_{pofs}(n))dt\end{bmatrix} \tag{3}$$

$$\begin{bmatrix}c_y\\ s_y\end{bmatrix} = \begin{bmatrix}cos(\theta_y(n))\\ sin(\theta_y(n))\end{bmatrix} \tag{4}$$

$$\begin{bmatrix}V_x(n)\\ V_y(n)\end{bmatrix} = \begin{bmatrix}V_x(n-1)+d_{vx}(n)c_y+d_{vy}(n)s_y\\ V_y(n-1)+d_{vx}(n)s_y+d_{vy}(n)c_y\end{bmatrix} \tag{5}$$

where $\theta_{rofs}(n)$ and $\theta_{pofs}(n)$ are offsets that could be estimated over time to compensate for misalignment of the accelerometer. Again, the singularity when the roll or pitch angle $[\theta_r(n), \theta_p(n)]^T$ are 90° is not an issue because these angles are limited at 45°. This is then used to update the position $[P_x(n), P_y(n)]^T$:

$$\begin{bmatrix}P_x(n)\\ P_y(n)\end{bmatrix} = \begin{bmatrix}P_x(n-1)+V_xdt\\ P_y(n-1)+V_ydt\end{bmatrix} \tag{6}$$

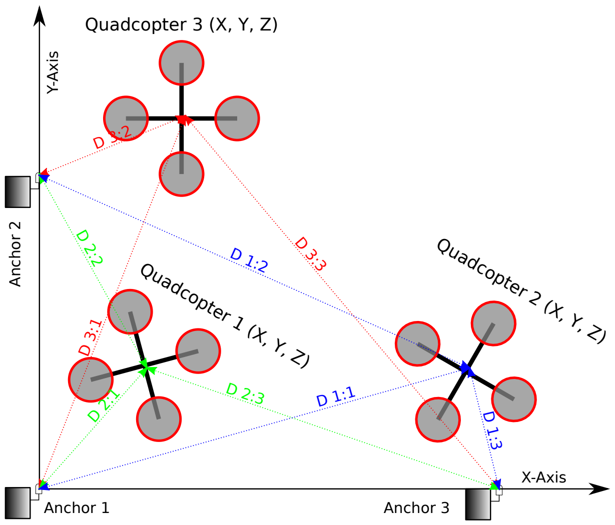


Figure 2: The ultrasound-localization system with anchors. The measured distances from the copters to the anchors are marked D a:b.

As the velocity integration drift is unbounded even when there is a small offset on the attitude estimation, the anchor distance measurements have to be used to estimate the velocity drift in addition to the roll and pitch error. For the anchor corrections, which arrive at a lower rate, we first compute the difference between the expected distance to the anchor from the dead-reckoning and the measured distance to the anchor:

$$\begin{bmatrix} d_{ax}(n) \\ d_{ay}(n) \\ d_{az}(n) \end{bmatrix} = \begin{bmatrix} P_x(n-1) - P_{x,anchor} \\ P_y(n-1) - P_{y,anchor} \\ P_z(n-1) - P_{z,anchor} \end{bmatrix} \tag{7}$$

$$d_a(n) = \sqrt{d_{ax}(n)^2 + d_{ay}(n)^2 + d_{az}(n)^2} \tag{8}$$

$$err(n) = d_a(n) - d_{measured}(n) \tag{9}$$

$$F_c(n) = \frac{err(n)}{d_a(n)} \tag{10}$$

where $[P_x(n), P_y(n), P_z(n)]^T$ is the position of the copter, $[P_{x,anchor}, P_{y,anchor}, P_{z,anchor}]^T$ is the position of the anchor this measurement came from and $[d_{ax}(n), d_{ay}(n), d_{az}(n)]^T$ is the difference between them. Further, $d_a(n)$ is the magnitude of the calculated difference, $d_{measured}(n)$ is the measured magnitude, $err(n)$ is the difference between the calculated and the measured magnitude and $F_c(n)$ is a factor that is used in later calculations for correction.

At this point, if the error is larger than a certain threshold, we discard this measurement and lower the position quality because something is likely to be wrong. If too many consecutive measurements have a large error, we stop discarding and start using them in case this is the initial position correction at start-up.

Next, the position differences $[d_{ax}(n), d_{ay}(n), d_{az}(n)]^T$ are used to correct the current position and the velocity error where we compute proportional and derivative parts, $[P_{xpos}(n), P_{ypos}(n)]^T$ and $[D_{xpos}(n), D_{ypos}(n)]^T$, on the position error. The gain components in the following equations $(G_{p,vel}, G_{p,pos}, G_{d,pos})$ were derived experimentally and the simulation presented in Section 3 has been an important aid for doing that.

$$\begin{bmatrix} P_{xpos}(n) \\ P_{ypos}(n) \end{bmatrix} = \begin{bmatrix} d_{ax}(n)F_c G_{p,pos} \\ d_{ay}(n)F_c G_{p,pos} \end{bmatrix} \tag{11}$$

$$\begin{bmatrix} D_{xpos}(n) \\ D_{ypos}(n) \end{bmatrix} = \begin{bmatrix} (d_{ax}(n)F_c - d_{ax}(n-1)F_c)G_{d,pos} \\ (d_{ay}(n)F_c - d_{ay}(n-1)F_c)G_{d,pos} \end{bmatrix} \tag{12}$$

Then, apply this to the position:

$$\begin{bmatrix} P_x(n) \\ P_y(n) \end{bmatrix} = \begin{bmatrix} P_x(n-1) + P_{xpos}(n) + D_{xpos}(n) \\ P_y(n-1) + P_{ypos}(n) + D_{ypos}(n) \end{bmatrix} \tag{13}$$

Updating the velocity state $[V_x(n), V_y(n)]^T$ is done in a similar way:

$$\begin{bmatrix} V_x(n) \\ V_y(n) \end{bmatrix} = \begin{bmatrix} V_x(n-1) + d_{ax}F_c(n)G_{p,vel} \\ V_y(n-1) + d_{ay}F_c(n)G_{p,vel} \end{bmatrix} \tag{14}$$

2.1 Collision Avoidance

Collision avoidance is done by placing risk contours around copters and static objects from the perspective of every copter, and steering away if the risk contours overlap with the comfort zone of the copter. This means that the risk contours are not a global state, but different from every copter's perspective based on its relative velocity to the object and when the positions of other copters were last received. The comfort zone is represented as a circle placed around the quadcopter with a radius that is calculated based on the confidence of the position estimation.

The risk contours are two-dimensional and represented as ellipses with width, height and rotation $[W_E, H_E, \theta_E]^T$, and they are sized and rotated based on the squared relative velocity vector to the copters/objects they surround. To share knowledge about the position of all copters, they broadcast this information one at a time to everyone else. When a copter receives a position update from another copter, it will update its Local Dynamic Map (LDM) (which contains the positions of all other known copters and the surrounding terrain) with this information. Between the position updates, the risk contours around other copters will be moved and reshaped based on the velocity that the other copters had when their position was last received. When an overlap between the comfort zone of a copter and a risk contour occurs, the collision-avoidance mechanism will take over control and steer away from the overlapping risk contour in the opposing direction. If there are several overlaps at the same time, a vector will be calculated from a weighted sum of all overlapping risk contours and their relative direction, and used to steer away from the collision.

2.2 Realistic Fault Models

On the hardware quadcopter platform, we have observed a number of fault sources that we use to derive realistic fault models. A list of where they originate from and how they affect the equations is given below:

- **Accelerometer misalignment**. The accelerometer provides the absolute reference gravity vector that points towards the ground, and if it is misaligned, the position estimation will be affected. This can be modelled by adding offset faults to $[\theta_r, \theta_p]^T$ in Equation 3. This fault will not change over time.

- **Air movement**. When flying outdoors, close to other quadcopters or close to objects, air movement and turbulences affect the localization. Since we have not included that in the model, it will affect the copters as accelerations. Similar to accelerometer misalignment, this can be modelled by adding offset faults to $[\theta_r, \theta_p]^T$ in Equation 3. Compared to the accelerometer misalignment fault, this fault changes more and faster over time. With FaultCheck, it can be added as a second fault on the same probes as the accelerometer faults.

- **Gyroscope drift**. A MEMS-gyro will drift over time [20] and affect the localization. This fault is also similar to the accelerometer misalignment fault and can be injected on $[\theta_r, \theta_p]^T$ in Equation 3. The drift is not constant like the accelerometer misalignment fault, but it changes slower than the air movement faults.

- **Gyroscope gain errors**. If the gain is not perfectly calibrated on the gyroscope, the position will drift while the quadcopter is moving. This can be modelled by adding amplification faults to $[\theta_r, \theta_p]^T$ in Equation 3. When the copter is perfectly leveled and not moving this fault will not have any effect.

- **Ranging reflections**. The localization does not always give perfect samples, and some of them can be much too long when the direct path is blocked to one anchor and a reflection is received. This can be modelled by adding a large random offset to the measured distance $d_{measured}(n)$ in Equation 9.

- **Anchor misplacement**. If one of the anchors is not placed where it is expected, the correction from it will not converge to the correct position over time. This can be modelled by adding a small offset to $[P_{x,anchor}, P_{y,anchor}, P_{z,anchor}]^T$ in Equation 7.

- **Communication faults**. If the radio channel is unreliable, communication faults, such as corrupted data, repeated packets and lost packets, can occur. This can be modelled by passing the packets sent between the copters through the communication channel of FaultCheck and injecting these communication faults on them.

The same variables in the equations are affected by different fault models that can be active simultaneously (e.g. the accelerometer can be misaligned at the same time as there is air movement and gyroscope drift). FaultCheck has a feature to inject simultaneous faults that can be controlled independently to the same variable with a single probe, which is useful when a SUT has fault models that behave in this way.

3 Quadcopter Simulator

Our quadcopter simulator is a library written in C++ with an interface where copters can be added, removed, or commanded to move. A block diagram of the simulator is shown in Figure 3. The block named *CopterSim* has a list of *CopterModels* and a list of line segments that represent static terrain. Every time dt CopterSim executes the state update function for each CopterModel and checks for collisions between all CopterModels and the static terrain. When a collision occurs, the simulation is halted and the position of the collision is reported. When a CopterModel is added to the simulation, CopterSim will upload the list of terrain to it and broadcast perceived position state messages from it to the other CopterModels and vice versa. This broadcast is done between all copters every communication time interval and the messages are passed through the communication channel of FaultCheck, where communication faults can be injected.

The *CopterModel* block runs the same source code for position and velocity estimation, shown in Equation 5 and 6, as the implementation on the hardware quadcopters. The angles $[\theta_r, \theta_p, \theta_y]^T$ are updated from the movement command with a similar response to that of the actual hardware, and the golden run (the true position and velocity state) is updated based on these angles. In addition to the true position state for each copter, CopterModel also updates the perceived position for them. For the perceived position, we have added FaultCheck probes to the various state variables, as described in Section 2.2, where faults can be injected. As long as no fault is activated the true and perceived position will be the same. As soon as we activate faults the positions will drift apart.

In order to compensate for faults and thus position drift, the perceived position has to be estimated using ultrasound sensor readings, as described in Section 2. CopterModel has a list of all anchors and simulates ultrasound-sensor readings based on the true position state and the anchor positions, with the same rate as they are received on the hardware copters. These readings are passed to the correction part of the position-estimation algorithm which corrects the position as described in Equation 7 - 14.

Every CopterModel block has a block named Intelligent Transportation System (ITS) station, which builds and updates a LDM that contains all other copters and their states as it receives messages from them. The ITS station also keeps track of the terrain (received from CopterSim) and runs the collision-avoidance mechanism, as described in Section 2.1, based on the LDM. Since the ITS-station operates on the perceived position of all copters, it is important that the position-estimations algorithm performs well when there are faults present

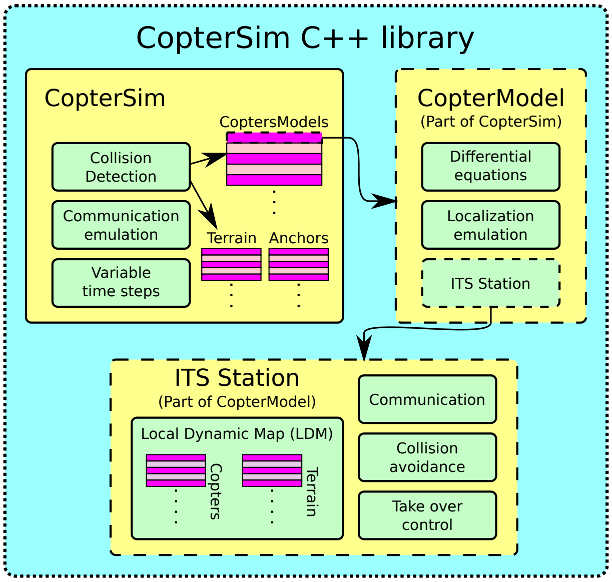


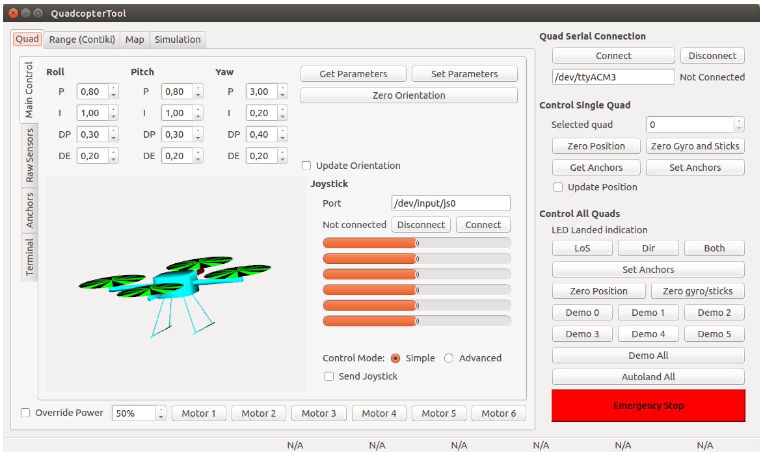
Figure 3: The simulator library. CopterSim has a list with all copters and checks for collisions between them and the terrain in every iteration. CopterModel handles the physical model of every copter and has an ITS-station that handles collision avoidance.

and that safety margins are large enough to cope with a slightly inaccurate perceived position.

3.1 QuadcopterTool

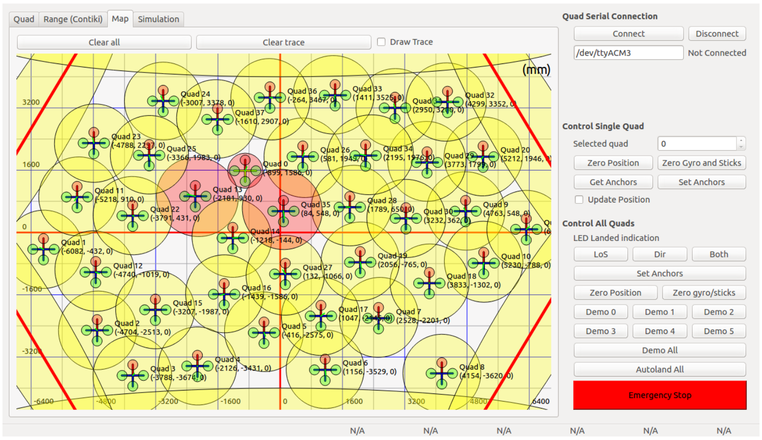
QuadcopterTool, as can be seen in Figure 4, is a Graphical User Interface (GUI) that is used to set up and control the hardware quadcopters. We have extended its interface and 2D-visualizations with the ability to visualize the state of all simulated quadcopters in CopterSim in real-time. CopterSim has the ability to send UDP-commands to QuadcopterTool with the states of all simulated quadcopters at the same time. CopterSim also sends the risk contours seen from a selected quadcopter to QuadcopterTool, which are represented as ellipses around other quadcopters and map line segments as seen in Figure 5. In Figure 6 it can be seen that the ellipses around the other copters from the perspective of Quad 4 are stretched because there is a relative velocity between them.

Even when running a simulation with many copters simultaneously using short time steps, the simulation and visualization is fast enough to run in real time. The simulation in Figure 5 has 40 quadcopters, an iteration time step of 5 ms and updates the map at 60 Hz, and can still run on a common laptop computer without dropping in frame rate.



Roll: P 0,80; I 1,00; DP 0,30; DE 0,20
Pitch: P 0,80; I 1,00; DP 0,30; DE 0,20
Yaw: P 3,00; I 0,20; DP 0,40; DE 0,20
Port /dev/input/js0
Not connected
Override Power 50%
/dev/ttyACM3 Not Connected
Selected quad 0

Figure 4: A screenshot of the main tab of QuadcopterTool. It is a GUI for controlling and visualizing the hardware quadcopters. We have updated the interface and the 2D-visualization of QuadcopterTool to receive UDP commands from CopterSim so that the simulation state, including all risk contours, can be visualized in real-time.



/dev/ttyACM3 Not Connected
Selected quad 0

Figure 5: A screenshot of the map in QuadcopterTool where a CopterSim simulation with many copters is visualized in real-time. The selected copter (with the smallest circle around it), has its comfort zone overlapping with the risk contours of the other copters (shown in red).

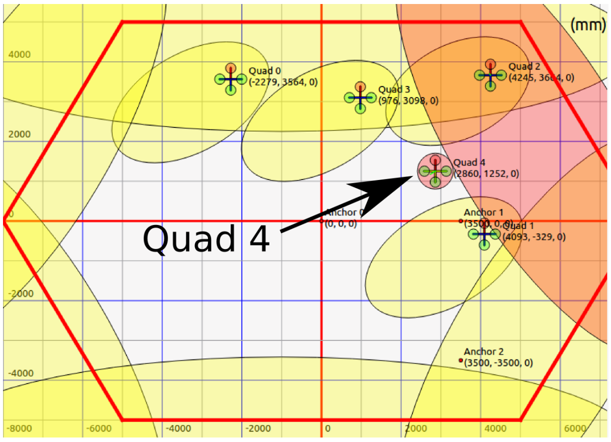


Figure 6: A screenshot of the risk contours from the perspective of Quad 4. The red contour is red because there is an overlap between the copter's own comfort zone (Quad 4) and the risk contour around the right upper wall.

3.2 CopterSimGUI

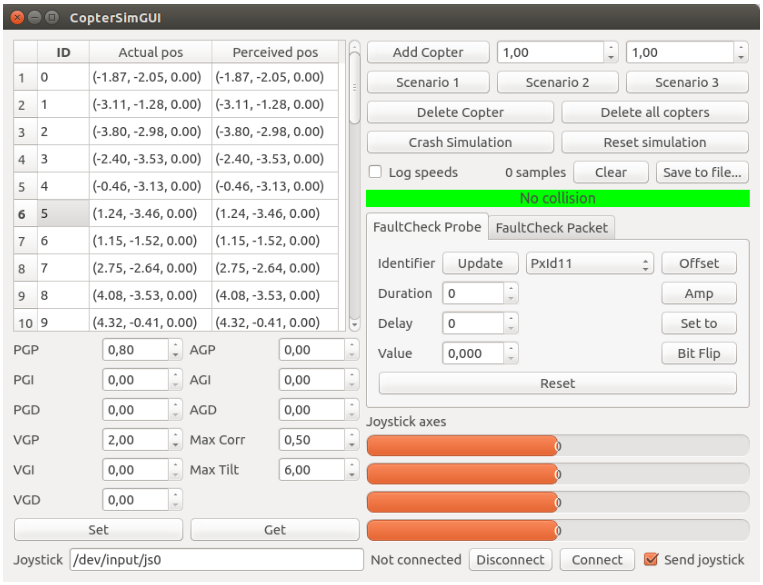
CopterSimGUI, seen in Figure 7, is a GUI that we developed to manually control the simulated quadcopters and inject faults using FaultCheck. It is useful when developing the simulator and to test a few FaultCheck probes at a time, but running a wide variety of simulations using it takes significantly more effort and time than using QuickCheck to automatically generate command sequences for CopterSim. Although generating simulations with QuickCheck is more effective than doing it manually using CopterSimGUI, CopterSimGUI is still useful during the development of the simulator since it is convenient to have the ability to test one feature at a time as they are added.

4 Testing CopterSim with our Testing Platform

The setup of our testing platform can be seen in Figure 8. CopterSim and QuadcopterTool are the SUT, and FaultCheck together with QuickCheck is our testing platform. QuickCheck sends steering commands to the CopterSim library and FI-commands to FaultCheck. How these commands are generated and behave is controlled by the model for QuickCheck.

4.1 QuickCheck Model

We have created a model for QuickCheck that sends commands to the simulator where we add a random number of copters at random non-overlapping positions and run commands while checking the property that they do not collide. These randomly-generated commands can either be steering commands for the copters, or fault-injection commands passed to FaultCheck.



	ID	Actual pos	Perceived pos
1	0	(-1.87, -2.05, 0.00)	(-1.87, -2.05, 0.00)
2	1	(-3.11, -1.28, 0.00)	(-3.11, -1.28, 0.00)
3	2	(-3.80, -2.98, 0.00)	(-3.80, -2.98, 0.00)
4	3	(-2.40, -3.53, 0.00)	(-2.40, -3.53, 0.00)
5	4	(-0.46, -3.13, 0.00)	(-0.46, -3.13, 0.00)
6	5	(1.24, -3.46, 0.00)	(1.24, -3.46, 0.00)
7	6	(1.15, -1.52, 0.00)	(1.15, -1.52, 0.00)
8	7	(2.75, -2.64, 0.00)	(2.75, -2.64, 0.00)
9	8	(4.08, -3.53, 0.00)	(4.08, -3.53, 0.00)
10	9	(4.32, -0.41, 0.00)	(4.32, -0.41, 0.00)

PGP	0,80	AGP	0,00
PGI	0,00	AGI	0,00
PGD	0,00	AGD	0,00
VGP	2,00	Max Corr	0,50
VGI	0,00	Max Tilt	6,00
VGD	0,00		

1,00 1,00

☐ Log speeds 0 samples

No collision

Identifier PxId11
Duration 0
Delay 0
Value 0,000

Joystick /dev/input/js0 Not connected ☑ Send joystick

Figure 7: CopterSimGUI is a simple GUI that we have developed for manual testing. It can be used to manually control the quadcopters and inject faults using FaultCheck.

The steering command has the parameters 1) which copter to command, randomly chosen from all the copters present in the simulation, 2) the roll output, randomly chosen between ±15°, 3) the pitch output, randomly chosen between ±15°, and 4) the yaw rate output, randomly chosen between ±90° per second. The only precondition is that the previous command is not a steering command, since this does not make any sense without having iterations between them. Further, the fault injection commands have the parameters 1) which copter to affect, randomly chosen from all the copters present in the simulation, and 2) the fault type, randomly chosen from the fault models described in Section 2.2.

All fault injection commands look similar and are essentially two different types of calls to FaultCheck. The first type is to the probing interface, and looks like the following:

```
%% Position offset fault
fault_pos_offset_pre(S, [Id, _OffX, _OffY]) ->
    length(S#area.faults) < ?MAX_FAULTS andalso
  not lists:member({pos_offset, Id}, S#area.faults).

fault_pos_offset_args(S) ->
    ?LET(Copter, elements(S#area.copters), [Copter#copter.id,
                choose(-?MAX_POS_OFFSET, ?MAX_POS_OFFSET)
,
                choose(-?MAX_POS_OFFSET, ?MAX_POS_OFFSET)
```

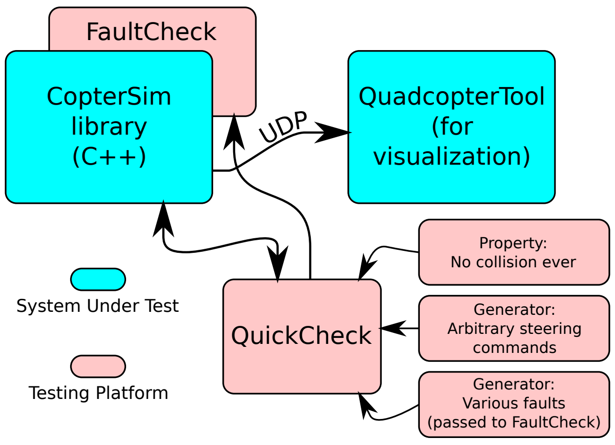

FaultCheck
CopterSim library (C++)
UDP
QuadcopterTool (for visualization)
Property: No collision ever
Generator: Arbitrary steering commands
Generator: Various faults (passed to FaultCheck)
QuickCheck
System Under Test
Testing Platform

Figure 8: Our testing platform (FaultCheck, QuickCheck) connected to our SUT (CopterSim, QuadcopterTool).

```
]).

fault_pos_offset_pre(S) ->
    S#area.copters /= [].

fault_pos_offset(Id, OffX, OffY) ->
    CStrX = "CopterOffsetXId" ++ integer_to_list(Id),
    CStrY = "CopterOffsetYId" ++ integer_to_list(Id),
    c_call:faultcheck_addFaultOffset(CStrX, OffX / 1000),
    c_call:faultcheck_addFaultOffset(CStrY, OffY / 1000).

fault_pos_offset_next(S, _, [Id, _OffX, _OffY]) ->
    S#area{faults = S#area.faults ++ [{pos_offset, Id}], prev_cmd =
fault}.
```

the other type is to the communication channel interface, and it looks in a similar way.

In both cases a precondition is that there are no more than MAX_FAULTS simultaneous faults and that not the same fault with the same parameters is already present. For example, having two position offsets on the same anchor will have the same result as having a single offset on it with the sum of both offsets. The reason that we limit the maximum number of simultaneous faults is that it is difficult to isolate the problem in a long test sequence with many faults.

There is also a command to run the simulation for a certain amount of time, named *iterate*. This command will instruct the simulator to run for a chosen amount of milliseconds. It is possible to make the iterations longer and run for less iterations or the other way around, depending on whether simulation speed or accuracy is more important. A constant is used to tell the simulator how often the copters are allowed to communicate with each other. By setting another constant, the simulator will send the simulation state to QuadcopterTool (see Figure 8). This will make the tests run much slower, but the test sequences are visualized while the tests are running, which can be useful for debugging. The *iterate* command is the only command with a postcondition, which is that no collision has occurred.

4.2 FaultCheck Integration

The integration of FaultCheck into this system required the steps 1) linking to the FaultCheck library in the build system of CopterSim 2) probing the code of CopterSim both with the communication channel and probing parts of FaultCheck and 3) linking to FaultCheck when starting the C code from Erlang using QuickCheck.

The probes are added to the *CopterModel* class code like the following:

```
// Inject ranging fault
const QString fcStr = QString().sprintf("RangeId%dAnch%d",
mItsStation.getId(), anch_int.id);
faultcheck_injectFaultDouble(fcStr.toLocal8Bit().data(), &
anchor_distance);
```

In this example, the string *fcStr* is the identifier, generated from the copter ID and anchor number, that can be used by QuickCheck to inject a fault here. Together with the identifier, a pointer to the variable *anchor_distance* is passed to FaultCheck. FaultCheck keeps track of all such probes and has list of fault models on each one them. When the fault models for the probes should be active and for how long is also handled by FaultCheck. As explained throughout the paper, FI is only done on the perceived positions of the simulated quadcopters and on the communication between them.

Where packets are sent between the simulated copters, they are passed through the communication channel of FaultCheck, which is simple to implement in the source code of CopterSim. FaultCheck handles buffering (for delay faults), modification (for corruption faults) and repetition (for repetition faults) of the packets.

The total amount of source code for the probes of FaultCheck in the simulator is about 15 lines, which is a small overhead for the integration.

5 Visualizing Test Sequences and Improving the System

When generating tests with QuickCheck, every time a test fails a sequence of the generated commands is printed. Since the state of the system is complex and difficult to see from only looking at the commands, we had to find a way to visualize what the command sequence actually meant. One way to do that would be to send the state of CopterSim to QuadcopterTool after each *iterate* command so that the position of all quadcopters could be seen in

QuadcopterTool (see Section 3). However, the problem with this would be that the iterate commands tend to be quite long (up to several seconds), hence the movement of the quadcopters cannot be followed smoothly until the collision.

One way to get a smooth replay of the command sequence is to split the iterate commands into several short parts and send the state to QuadcopterTool after each such part and then put the replay thread to sleep for the duration of the part. As the sleeping time between each part of the iterate command can be varied, this can be used to change the playback speed of the command sequence. By playing the commands slower, more details about the collision can be observed.

Because CopterSim is restarted every time the commands are replayed, modifications can be made to the code between the replays. This way, system parameters and the collision-avoidance mechanism can be adjusted and tested over and over again until the quadcopter system can handle the encountered faults.

5.1 Handling Faults in the Quadcopter System

While running auto-generated tests, we discovered several scenarios that led to collisions while faults were injected. This is a summary of type of changes we made to the quadcopter system to deal with those faults:

- Making the quadcopters comfort zone bigger. This will cause them to keep larger safety distance and thus make them less sensitive to position estimation errors.

- Communicating more often. One way to deal with lost and corrupted packets between the copters is to communicate more often to compensate for that.

- Placing the anchors that the copters measure their distance from more accurately. Setting up the localization system correctly helps to improve the position estimation.

- Adjusting the position-estimation algorithm. Having a more accurate position while there are faults present will decrease the probability of collisions.

- Filtering out outliers in the position-estimation algorithm. When an ultrasound sensor reports a distance with a random offset, corresponding to a bounce and not the direct path, the value can be compared to the previous one and ignored if it differs too much.

It can also be concluded that if we still get collisions for certain combinations of faults even though we have used the countermeasures above, we have to find a way to make sure pre-runtime that a combination and/or intensity of faults that cannot be handled is not encountered.

6 Conclusions

We have created a simulation environment, based on a hardware quadcopter platform, where we can auto-generate tests and inject faults for many copters simultaneously, making it possible to scale up the tests beyond what the physical hardware allows. We have shown a practical example on how FaultCheck and QuickCheck can be used together to effectively enhance the copter's collision-avoidance mechanism and adjust system parameters (e.g. communication rate and safety margins) to reduce the risk of collisions under realistic conditions. Additionally we have shown how to visualize the state of the SUT during a sequence of QuickCheck commands for CopterSim and FaultCheck in an intuitive way, while providing the possibility to replay the visualization while adjusting the SUT.

The overhead from integrating FaultCheck into the simulator, including generating the perceived position state for each copter, is only 2 % of the total amount of source code of the CopterSim library. Additionally, our QuickCheck model consists of 320 lines of Erlang code. Although the amount of code is not an accurate measure to compare software, it shows that not a significant amount of extra effort was required to use our testing platform on our quadcopter simulator.

Even though the quadcopter simulator is a complex SUT, using our testing platform on it was straight forward. It provided many advantages such as the ability to test the collision-avoidance mechanism and the position-estimation algorithm together under realistic conditions. This gives us confidence that our testing platform is a useful aid when developing and testing a wide range of complex systems from different domains where faults have to be handled effectively during operation of the system.

7 Acknowledgement

This research has been funded through the PROWESS EU project (Grant agreement no: 317820), the KARYON EU project (Grant agreement no: 288195) and through EISIGS (grants from the Knowledge Foundation).

References

[1] R. K. Iyer. "Experimental Evaluation". In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing*. FTCS'95. Pasadena, California: IEEE Computer Society, 1995, pp. 115–132.

[2] 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*. 1994, pp. 66–75.

[3] V. Sieh, O. Tschache, and F. Balbach. "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions". In: *Proceedings of the Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*. 1997, pp. 32–36.

[4] K. K. Goswami, R. K. Iyer, and L. Young. ”DEPEND: A Simulation-Based Environment for System Level Dependability Analysis”. In: *IEEE Transactions on Computers* 46 (1997), pp. 60–74.

[5] 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.* 2007, pp. 1–9.

[6] R. Svenningsson, H. Eriksson, J. Vinter, and M. Törngren. ”Model-Implemented Fault Injection for Hardware Fault Simulation”. In: *Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa).* Oct. 2010, pp. 31–36.

[7] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren. ”MODIFI: a Model-Implemented Fault Injection tool”. In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security.* SAFECOMP'10. Vienna, Austria: Springer-Verlag, 2010, pp. 210–222.

[8] A. Joshi and M. Heimdahl. ”Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier”. In: *SAFECOMP.* Vol. 3688. LNCS. 2005, p. 122.

[9] M. Hiller. ”PROPANE: An Environment for Examining the Propagation of Errors in Software”. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM Press, 2002, pp. 81–85.

[10] 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.* ES4CPS '14. Dresden, Germany: ACM, 2014, 1:1–1:8.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell. ”Fault Injection for Dependability Validation: A Methodology and Some Applications”. In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 166–182.

[12] H. Madeira, M. Rela, F. Moreira, and J. Silva. ”RIFLE: A General Purpose Pin-Level Fault Injector”. In: *Dependable Computing — EDCC-1.* Ed. by K. Echtle, D. Hammer, and D. Powell. Vol. 852. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 197–216.

[13] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. ”Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms”. In: *IEEE Micro* 14.1 (1994), pp. 8–23.

[14] 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.* 1998, pp. 284–293.

[15] 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.* 2001, pp. 83–88.

[16] D. Skarin, R. Barbosa, and J. Karlsson. ”GOOFI-2: A tool for experimental dependability assessment”. In: *International Conference on Dependable Systems and Networks (DSN).* June 2010, pp. 557–562.

[17] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects*. Ed. by F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 250–271.

[18] T. Arts, J. Hughes, J. Johansson, and U. Wiger. ”Testing Telecoms Software with Quviq QuickCheck”. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang*. Portland, Oregon: ACM Press, 2006.

[19] S. Madgwick, A. J. L. Harrison, and R. Vaidyanathan. ”Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm”. In: *IEEE International Conference on Rehabilitation Robotics (ICORR)*. 2011, pp. 1–7.

[20] J. D., C. Gerdtman, and M. Linden. ”Signal processing algorithms for temperauture drift in a MEMS-gyro-based head mouse”. In: *International Conference on Systems, Signals and Image Processing (IWSSIP)*. May 2014, pp. 123–126.

Paper IV

Accurate Positioning of Bicycles for Improved Safety

Benjamin Vedder, Jonny Vinter and Magnus Jonsson

Published in Proceedings of the IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, USA, 2018

Abstract

Cyclists are not well protected in accidents with other road users, and there are few active safety systems available for bicycles. In this study we have evaluated the use of inexpensive Real-Time Kinematic Satellite Navigation (RTK-SN) receivers with multiple satellite constellations together with dead reckoning for accurate positioning of bicycles to enable active safety functions such as collision warnings. This is a continuation of previous work were we concluded that RTK-SN alone is not sufficient in moderately dense urban areas as buildings and other obstructions degrade the performance of RTK-SN significantly. In this work we have added odometry to the positioning system as well as extending RTK-SN with multiple satellite constellations to deal with situations where the view of the sky is poor and thus fewer satellites are in view. To verify the performance of the positioning system we have used Ultra-Wideband radios as an independent positioning system to compare against while testing during poor conditions for RTK-SN. We were able to verify that adding dead reckoning and multiple satellite constellations improves the performance significantly under poor conditions and makes the positioning system more useful for active safety systems.

1 Introduction

Cyclists are vulnerable road users and today there are few active safety systems available for them. In Sweden Studies have shown that among all types of bicycle accidents, the second most common type of accident is between cyclists and motor vehicles [1]. The same study also shows that 69 % of accidents that were fatal for the cyclist are collisions with motor vehicles. Given that travelling by bicycle is getting more common in urban environments, which is where 90 % of serious bicycle accidents occur [1], there is a strong incentive to improve bicycle safety.

If the position of a bicycle can be measured with precision greater than 0.5 m this information can be used for active safety systems that, for example, warn other road users when a bicycle approaches their trajectory in such a way that a collision may occur. Another example is to equip bicycle helmets with Augumented Reality (AR) devices that warn the cyclist about other road users that are at risk of a collision with them or about upcoming dangers such as damaged or slippery roads.

For the proposed positioning system to be practical it must not only fulfil the accuracy requirement, it must also be inexpensive, have low power consumption and not occupy too much space. In previous work [2] we have evaluated the use of inexpensive Real-Time Kinematic Satellite Navigation (RTK-SN) for positioning under static and dynamic conditions and concluded that it works well under good conditions, but urban environments with occasionally poor view of the sky degrades the performance significantly. In this study we have improved the positioning system by using more satellite constellations and by introducing dead reckoning using odometry and inertial navigation. For evaluating the performance we have developed a separate positioning system that uses Ultra-Wide Band (UWB) radios as a reference, which is unaffected by the conditions in our experiments that degrade the performance of RTK-SN. We observed significant improvements of the performance under poor conditions for RTK-SN when using more satellite constellations and when incorporating dead reckoning. The proposed positioning system is integrated on a compact Printed Circuit Board (PCB) developed by us that externally only requires an antenna for RTK-SN, odometry input and a power supply.

The remainder of this paper is organized as follows: in Section 2 we report related work in the area and in Section 3 we present our proposed implementation. Section 4 describes our evaluation method, while Section 4 presents our experiment results. In Section 6 we present our conclusions from this work.

2 Related Work

Active safety systems, that avoid accidents for cyclists, are of major importance and more research in the area is needed. Due to new technologies and low-cost consumer electronics, the market for active safety systems for cyclists is foreseen, by the authors, to grow in the near future. One strategy that has been used according to the literature is to equip bicycles with sensors to detect the presence and motion of other road users, and warn the cyclist locally or the other road user with sound signals when possible collisions are predicted. Examples of such attempts involve equipping the bicycle with radar [3], sonar and laser sensors [4]. Another approach is to rely on communication between road users and notifying them when possible collisions are predicted so that they can prevent them [5, 6].

In an early study specialized tags with short range communication and Radio-frequency identification (RFID) beacons built into the infrastructure near intersections for positioning has been proposed to identify when cyclists are about to collide with cars [6]. These tags, carried by the cyclist and car driver, would warn them based on communication between them and the relative position knowledge from the beacons in the infrastructure. As smartphones have become more widespread in use, later studies have proposed to use them as a display [3] and even as a localization and communication device [5].

One of the later studies proposes to use a smartphone mounted to the handle bar of the bicycle [5]. Since the smartphone is the only equipment used in that study, it is used for both localization with the built in Global Navigation Satellite System (GNSS) receiver and for communication using the built in internet connection. The achieved accuracy for the positioning was around 5 m and the communication delay was around 500 ms. With driver intent inference [7] near intersections, delay compensation and map knowledge, they were able to predict possible collisions during turns in an intersection when a car crosses the path of a cyclist. However, the smartphone positioning was not accurate enough to determine which lane the bicycle was in, which is limiting in many situations. Further, the route where they conducted the experiments had relatively good view of the sky, which was a requirement as they used RTK-SN as a reference positioning system. They also concluded that the positioning performance of the smartphone degrades too much to be useful for collision prediction when worn in a pocket, meaning that the smartphone must be attached to the handle bar of the bicycle.

Among the above mentioned studies, our work is best compared against the study where a smartphone is mounted on the handle bar of a bicycle. The advantage of our proposed implementation is that the position accuracy is at least one order of magnitude better and that communication delay is significantly lower. This makes it possible to determine in which lane the cyclist and other vehicles are, and collisions can be predicted without having access to an accurate and detailed map of the environment that the position has to be matched against. The cost of our proposed implementation is similar to that of a smartphone, but the advantage of using a smartphone is that many people already have access to one. Compared to the studies where sensors are mounted on the bicycle our solution has the advantage that less hardware is required and that the collision-prediction algorithm has access to more data earlier, but the disadvantage is that it only works together with road users that use a similar system that can communicate with our system.

3 Proposed Implementation

We have made an attempt at demonstrating a realistic implementation of our positioning system on a bicycle. Fig. 1 shows a rendering of a custom PCB that we have developed, containing an Inertial Measurement Unit (IMU), a RTK-SN module, Controller Area Network (CAN)-bus interface, two radios for communication and a cortex M4 microcontroller that performs sensor fusion. The size of the PCB is 55 x 98 mm, but this could be reduced significantly if the unused extension connectors are omitted. Connecting this PCB to an antenna for RTK-SN and a power supply is enough to get position information at a rate of 5 Hz with an accuracy of 10 cm or better, when the view of the sky is good enough to track at least 5 GPS and/or GLONASS satellites. If the PCB is provided with odometry data from a wheel sensor on the bicycle, the position update rate increases to 100 Hz and it is possible to operate under bad conditions for RTK-SN during short distances using dead reckoning from the odomotry data and the IMU alone. The power consumption of our PCB is around 200 mA without any power saving optimizations, which means that two 3 Ah 18650-size lithium-ion battery cells with a total weight of 100g can power it for more than 20 hours on one charge. When omitting odometry, the PCB, antenna and battery could be integrated in a bicycle helmet, possibly together with some type of AR display.

We have attached this PCB to an electric bicycle that we have access to. The reason for using our electric bicycle is that it has a motor controller that can provide odometry data over CAN-bus and power to the positioning PCB, but a regular bicycle with a wheel sensor and a small battery would work equally well. Fig. 2 shows the PCB mounted on a wood plate together with an UWB radio for distance measurement. The UWB radio is described further in Section 4. Fig. 3 shows the same wood plate attached to the battery box of our electric bicycle, where the RTK-SN antenna also can be seen on top of the plate. The

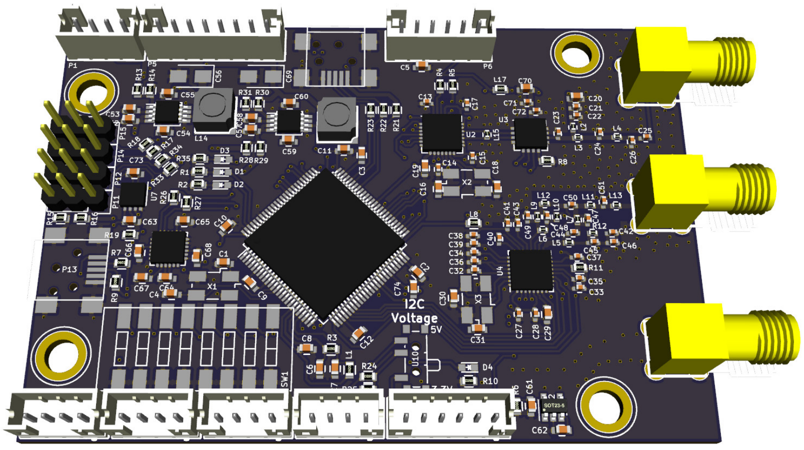


Figure 1: Rendering of our custom positioning PCB.

only connection from the plate is the white cable, which has both CAN-bus and power coming from the motor controller of the bicycle.

The two radio units on the PCB operate at different frequencies and data rates. One radio operates at around 450 MHz and is intended to receive a correction data stream for RTK-SN that is sent over distances of up to 10 km with a bandwidth of around 100 B/s. The other radio operates at 2.4 GHz and is intended for short range communication between road users with a bandwidth of 250 kbit/s. Compared to the hundreds of ms latency of an internet connection, this radio enables road users to communicate with a latency of 1 - 2 ms, similar to what the communication standard 802.11p achieves [8] and thus a realistic delay for future connected vehicles.

3.1 Real-Time Kinematic Satellite Navigation

The main positioning technology of our implementation is RTK-SN, which can achieve an accuracy of around 1 cm [9]. In addition to doing code measurement of the GNSS signals, RTK-SN works by also locking to the carrier phase of the signals to achieve significantly higher resolution. Compared to code measurements, the carrier-phase measurements do not provide means to measure the absolute distance to the satellites, but only to track changes in distance with high accuracy and resolution. Therefore RTK-SN requires the same raw measurements from a base station with a well known position that is within 10 km for achieving high accuracy. Those measurements from the base station are referred to as correction data, and they are required at a rate of approximately 1 Hz. The correction data from the public L1 signals of all GPS and GLONASS satellites visible on average can be transmitted in around 100 bytes of data using the RTCM3 transmission standard[1].

[1]http://www.rtcm.org/

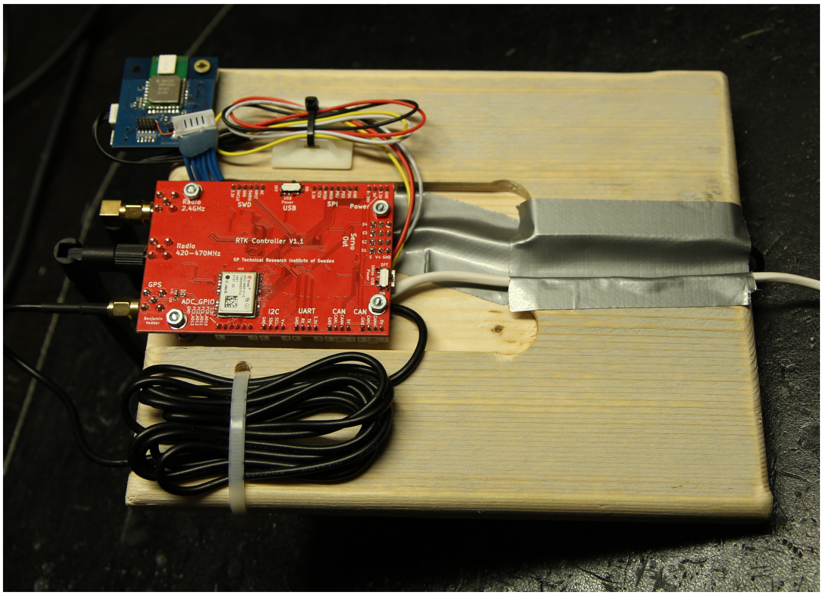


Figure 2: The positioning PCB and an UWB module mounted on a wood plate.

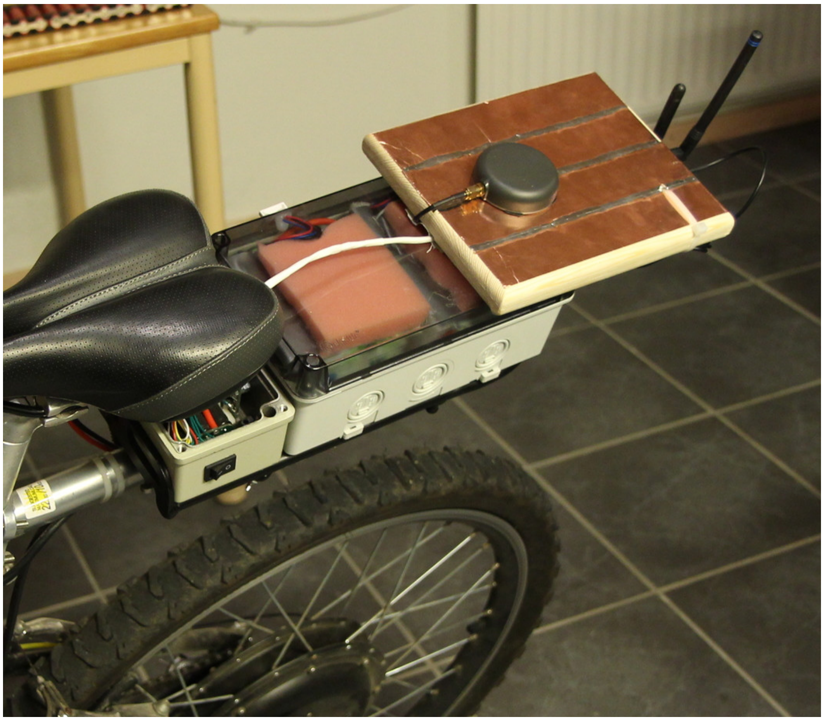


Figure 3: Bicycle equipped with our positioning system.

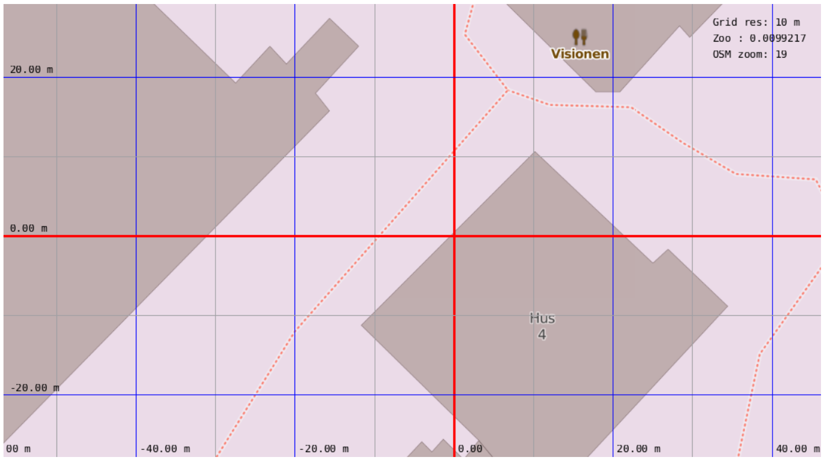
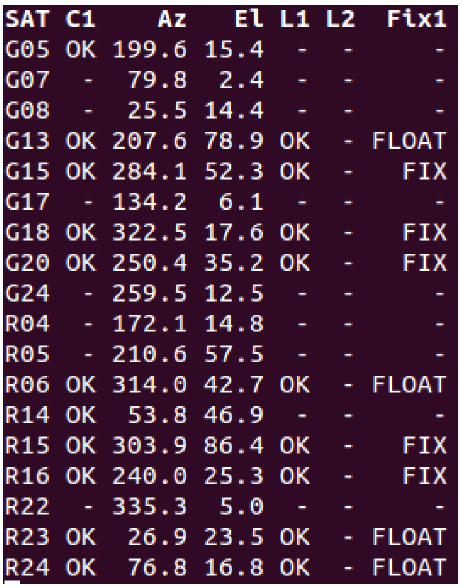


Figure 4: The red cross shows where the antenna was placed in a window of a building. The background rendering with buildings and walking trails is generated by OpenStreetMap [12].

When starting an RTK-SN system with the Ublox M8P[1] receivers we are using, it takes around one minute for the position to converge after locking to the signals from most visible satellites. This time is affected by how many satellites are in view and how good the used antenna is at rejecting reflections [10]. Every time less than 4 satellites are tracked the position fix is lost and the solution has to converge again, which is why dense urban environments are challenging for RTK-SN. Not only is the position information lost when too much of the sky is blocked, it also takes a relatively long time to converge after a lost position when enough, but still few, satellite signals can be tracked. Therefore, having access to more satellites is extremely important in urban environments. Compared to our previous work where we only used GPS satellites [2], we saw a significant improvement when also including GLONASS satellites when the view of the sky is poor. As an example, Fig. 4 shows the placement of a RTK-SN antenna in a window which is close to some trees and other buildings, and Fig. 5 shows the tracking output of the open source software package RTKLIB [11] from this placement. The first column shows a number starting with G for GPS satellites and a number starting with R for GLONASS satellites, and the last column shows FLOAT or FIX, depending on whether integer ambiguities have been resolved for that satellite, for satellites that are used in the current position solution. With both GLONASS and GPS satellites, RTKLIB was able to maintain a position solution from the tracked satellite signals during a whole day without problems, but when only using GPS satellites a position solution was achieved only a few times during the day lasting for 5 minutes or less every time.

[1]https://www.u-blox.com/en/product/neo-m8p-series



```
SAT C1    Az    El  L1 L2  Fix1
G05 OK 199.6 15.4  -  -      -
G07  -  79.8  2.4  -  -      -
G08  -  25.5 14.4  -  -      -
G13 OK 207.6 78.9 OK  - FLOAT
G15 OK 284.1 52.3 OK  -   FIX
G17  - 134.2  6.1  -  -      -
G18 OK 322.5 17.6 OK  -   FIX
G20 OK 250.4 35.2 OK  -   FIX
G24  - 259.5 12.5  -  -      -
R04  - 172.1 14.8  -  -      -
R05  - 210.6 57.5  -  -      -
R06 OK 314.0 42.7 OK  - FLOAT
R14 OK  53.8 46.9  -  -      -
R15 OK 303.9 86.4 OK  -   FIX
R16 OK 240.0 25.3 OK  -   FIX
R22  - 335.3  5.0  -  -      -
R23 OK  26.9 23.5 OK  - FLOAT
R24 OK  76.8 16.8 OK  - FLOAT
```

Figure 5: The visible satellites when the antenna was placed in the window.

3.2 Dead Reckoning

In order to get higher position update rate and to better deal with situations where the RTK-SN positioning solution is lost for short durations, we have implemented support for dead reckoning on our positioning system. Our implementation of dead reckoning works by using odometry data from the bicycle wheel rotation to determine how far the bicycle moves and yaw angle from the IMU to determine the direction of movement. When the wheel slip is low, which usually is the case for cyclists, the distance measurement has errors of less than 1 m after moving a distance of 100 m. The main source of error during dead reckoning is caused by the yaw estimation from the IMU data. Yaw information can be obtained from the IMU in two different ways: by integrating the angular velocity provided by the gyroscope and by measuring the direction of the magnetic field of the earth using the magnetometer.

The problem with using the gyroscope is that the yaw angle will drift over time as there is no absolute reference and small errors will accumulate [13]. Therefore the gyroscope angle has to be corrected with an absolute source, which either can be obtained from consecutive RTK-SN samples while moving, or from the magnetometer while standing still. The magnetometer does not have the problem of drift, but it suffers from offsets caused by metallic objects on the bicycle itself and in the environment where the bicycle moves. We have compensated for offsets on the bicycle itself by an ellipsoid fit and transformation method [14], but the offsets caused by the environment are more difficult to deal with as they cannot be predicted easily.

This means that the main error source of dead reckoning is the yaw angle estimation. The gyroscope alone gives a good approximation that drifts with about 0.5 degrees per minute in our experiments. While moving, the RTK-SN position samples keep this error well below one degree and situations where RTK-SN is unavailable for short durations can be dealt with, but when the bicycle is standing still this error accumulates without an upper bound. The magnetometer can be used for yaw correction even when standing still, but it causes yaw errors of over 10 degrees in some environments, which degrades the dead reckoning performance significantly. In Section 4 we show experiment results both with and without using the magnetometer for yaw correction and compare them.

4 Evaluation Setup

In order to evaluate the performance of our positioning system, we have developed a distance measurement PCB based on Decawave DWM1000 UWB radios [15] and a microcontroller which is unaffected by many error sources for RTK-SN. The DWM1000 radios have a high resolution clock and can send packets at precise times, and time stamp received packets precisely. This can be used to determine the Time of Flight (ToF) of the radio signals, which can be divided by the speed of light to calculate the distance between a sending and receiving radio. This measured time is very short compared to the time it takes to send the packets, so small clock speed differences between the radios have a large impact on the measurements. One way to deal with this problem is using a scheme called Two-Way Ranging (TWR), where three packets are sent between two radios, between which a distance is to be measured. This is done to remove the clock speed difference from the calculation [16]. By adding further compensation for distance-dependent signal attenuation which affects where the packets are time stamped by the hardware [17], around 10 cm of absolute accuracy can be achieved with the DWM1000 radios. Further, to reduce noise during the measurements, our implementation makes 10 measurements using TWR and takes the average of them for each distance sample.

One of these UWB PCBs was connected over CAN-bus to our positioning PCB and three other UWB PCBs were placed on tripods together with batteries as shown in Fig. 6 and used as anchors for the UWB positioning system. The distance between the bicycle and these anchors was then measured using the UWB radios and calculated using the known anchor position and the estimated position of the bicycle. By comparing the measured distance to the estimated distance from the bicycle an error was calculated.

It should be noted that RTK-SN under moderate to good conditions has better accuracy than the UWB distance measurement and that there are geometry conditions in our experiment, such as the assumption that the anchors and the bicycle are in a level plane, that have a negative impact on the accuracy. However, the accuracy of the UWB system is still sufficient to determine if the positioning system on the bicycle has sufficient performance for our application, and as it has independent error sources it can be used in conditions that are challenging for RTK-SN.

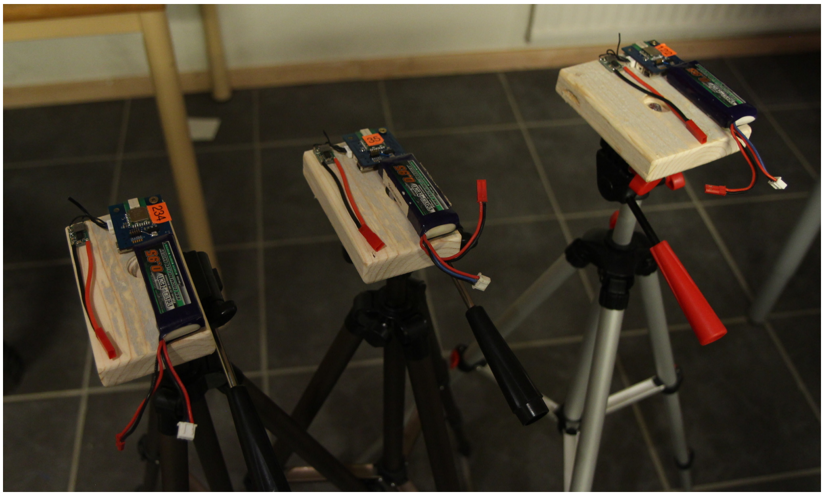


Figure 6: Three UWB PCBs placed on tripods together with batteries.

5 Results

Fig. 7 shows the three UWB anchors and the base station antenna for RTK-SN outside of a garage in a residential area. A portion of the sky is covered for the base station antenna and for the bicycle, so the conditions for RTK-SN are good but not optimal.

Notice the following: the figures with traces on maps in this section have two lines: a red line that represents the bicycle trace determined by sensor fusion between dead reckoning and RTK-SN; and a magenta line that represents the bicycle trace using RTK-SN alone. Under good conditions the lines are mostly overlapping and indistinguishable from each other, but under bad conditions there is a difference between them. The figures with distance deviation plots represent the difference between the calculated distance to the closest UWB anchor and the UWB distance measurement to the same anchor. The deviation between the UWB measurement and calculated distance to the anchor based on sensor fusion between dead reckoning and RTK-SN is the blue line, and deviation from RTK-SN alone is the red line. Errors shown in these plots are mostly caused by the UWB measurements under good conditions for RTK-SN, but in the last experiment where the bicycle is in the garage where the satellite signals are mostly blocked the UWB measurements have higher accuracy.

First we made a test cycling 20 m with a slight turn in the end under good conditions. The map traces are shown in Fig. 8 and the distance difference to the UWB measurements are shown in Fig. 9. The traces on the map in Fig. 8 are overlapping and the distance deviations in Fig. 9 are less than 35 cm the whole time.

We made another test under good conditions for RTK-SN with more turns, which is challenging when using the magnetometer for correcting the yaw angle because of its accuracy as mentioned in Section 3.2. Fig. 10 shows a cycling with

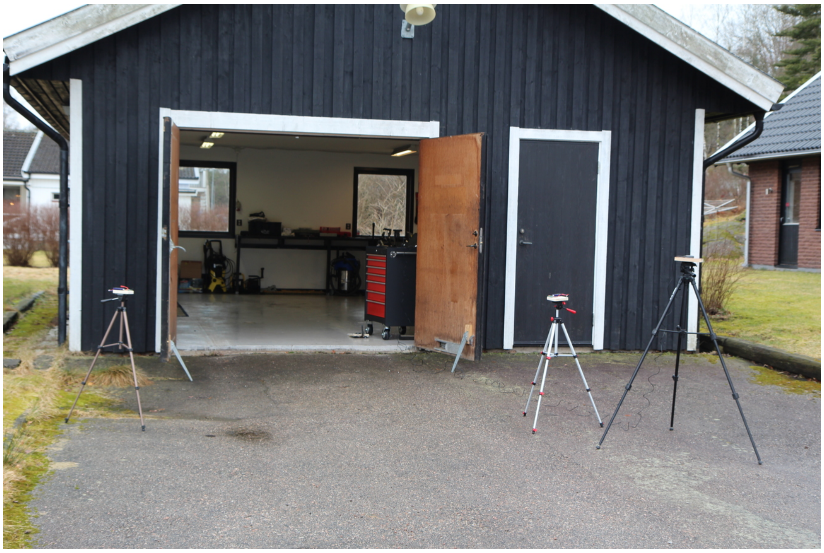


Figure 7: Two of the three UWB anchors and the base station antenna for RTK-SN, which is the higher tripod.

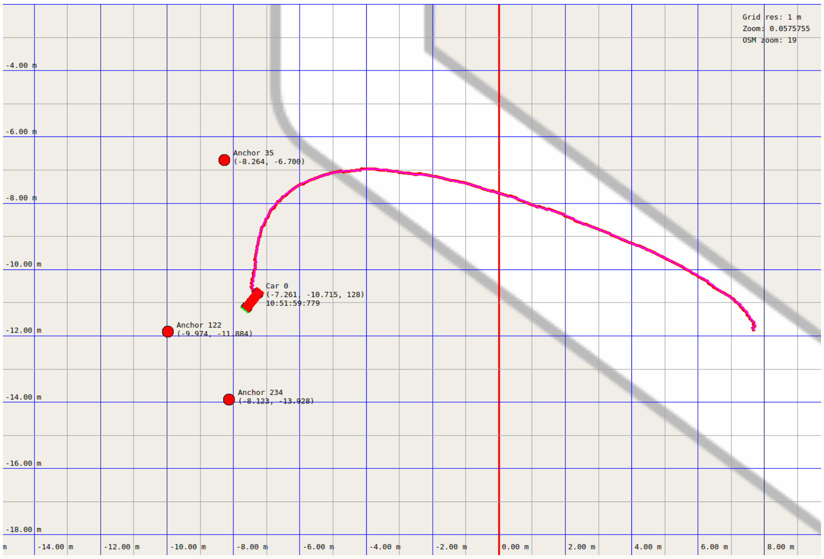


Figure 8: A 20 m cycling under good conditions.

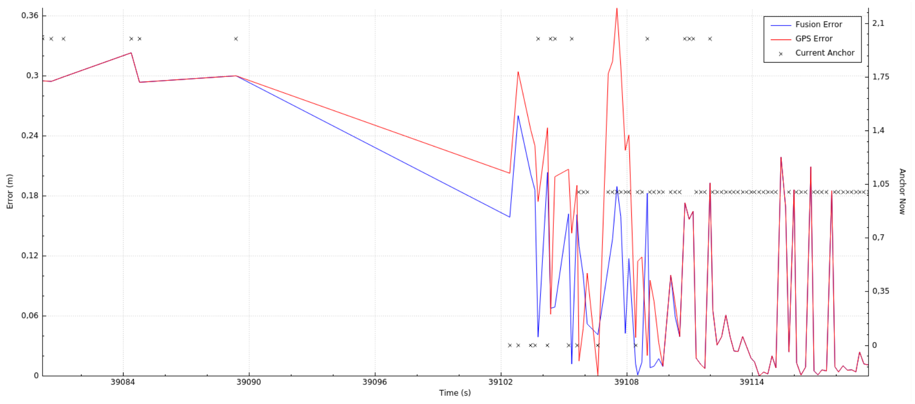


Figure 9: Distance deviations during the 20 m cycling under good conditions. The deviation is less than 35 cm for the whole cycling.

several turns where the magnetometer was used for yaw correction and Fig. 11 shows a cycling with several turns where RTK-SN was used for yaw correction. As can be seen, the cycling where RTK-SN was used for yaw correction has completely overlapping traces for the RTK-SN position and the sensor fusion position, while traces for when the magnetometer was used for yaw correction has some deviation between the traces close to turns. This is the expected outcome during movement, as described in Section 3.2.

Another test was made without using the magnetometer were part of the cycling was inside the garage. The trace for that cycling is shown in Fig. 12 and the distance deviations for that cycling are shown in Fig. 13. As can be seen on the traces in Fig. 12, when the bicycle is in the garage the trace with sensor fusion is stable and even shows the manoeuvre where the bicycle was pushed forth and back to turn around, while the trace with RTK-SN alone only jumps around while the bicycle is inside the garage. When the quality drop of RTK-SN was noticed by the sensor fusion filter inside the garage, the RTK-SN correction was dropped and only dead reckoning used. This can be seen in Fig. 13 in the middle where a deviation of 1 m was sustained for the sensor fusion position while the deviation for RTK-SN alone climbs to almost 3 m. At the end when the bicycle leaves the garage RTK-SN recovers and then it is included in the sensor fusion again where the deviations converge to the same value.

These experiments show that RTK-SN alone works sufficiently for accurate positioning of bicycles under moderate to good conditions, and that the addition of dead reckoning using odometry and an IMU helps in situations where the conditions for RTK-SN are bad.

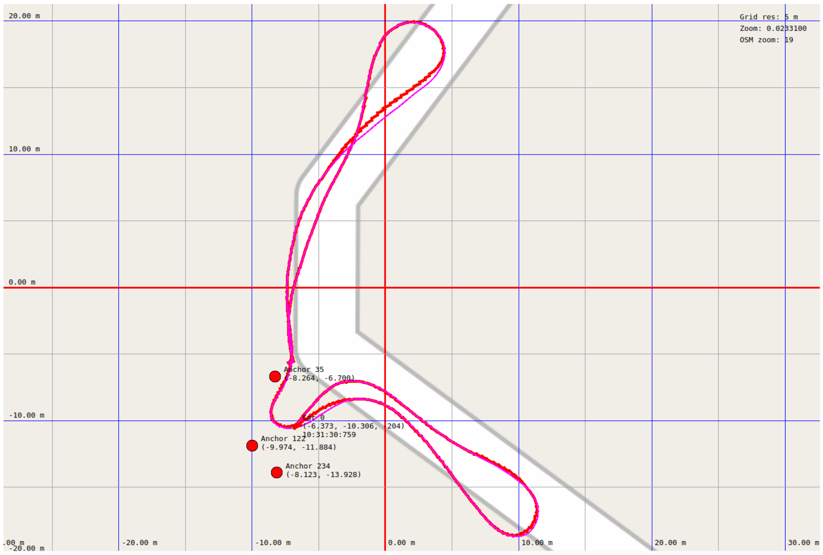


Figure 10: A cycling with some turns using the magnetometer for yaw correction.

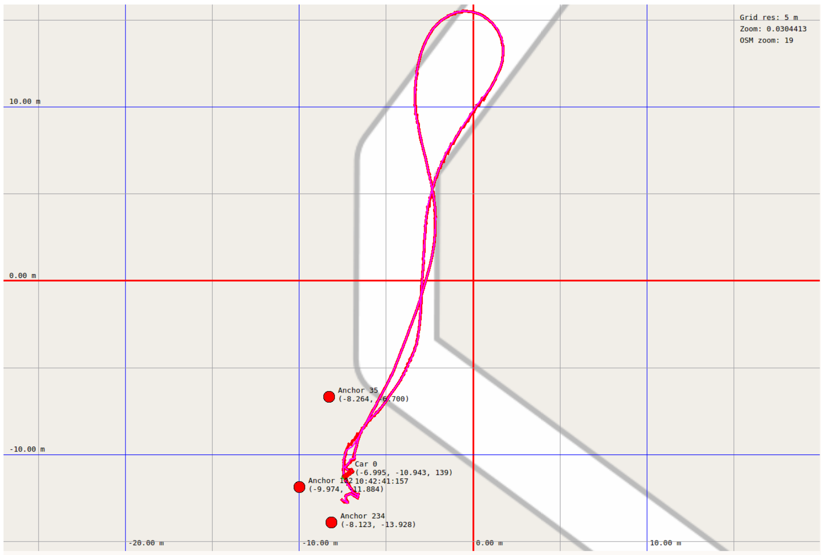


Figure 11: A cycling with some turns using RTK-SN for yaw correction.

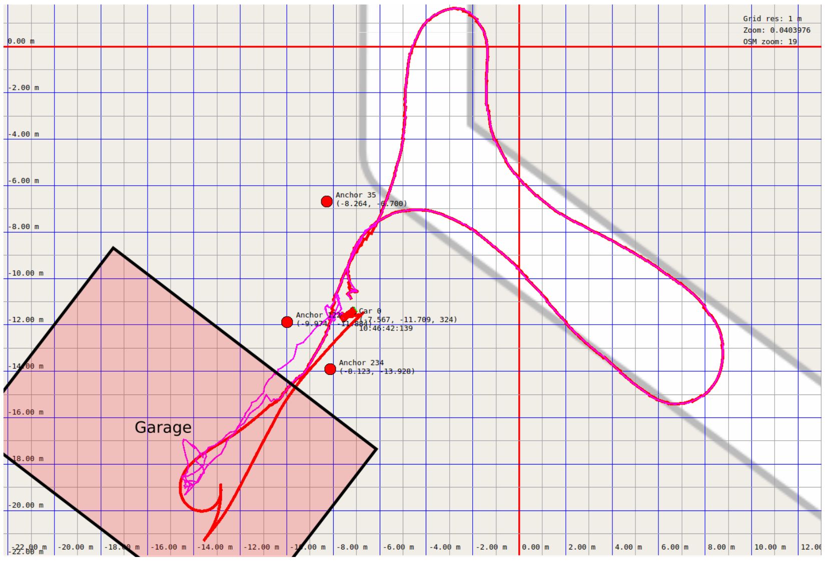

Grid res: 1 m
Zoom: 0.0403976
OSM zoom: 19
Anchor 35 (-8.264, -6.700)
Anchor 1
Car 0 (-7.567, -11.709, 324) 10:46:42:139
Anchor 234 (-8.123, -13.928)
Garage

Figure 12: A cycling that partly goes through the garage.

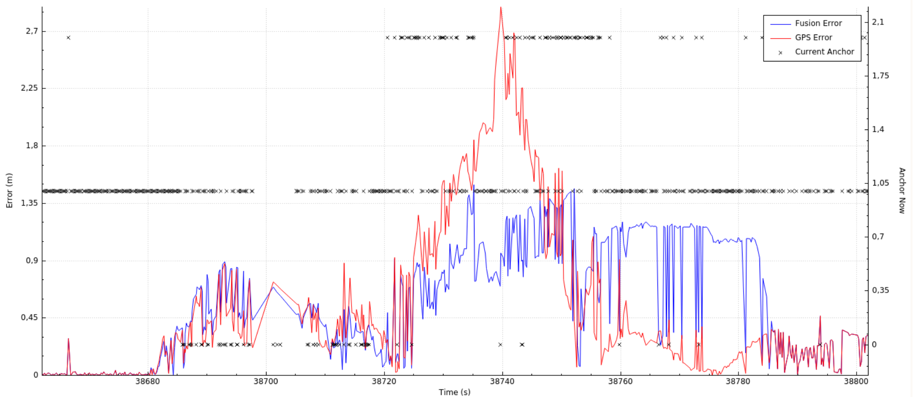


Figure 13: Distance deviations during the cycling that partly goes through the garage.

6 Conclusion

In this paper we have presented a novel positioning and communication system that can be used on bicycles for active safety functions. The system has roughly the same cost as a smartphone and one order of magnitude better position accuracy than the built in GNSS receiver of a smartphone. The delay when communicating with other road users is significantly lower compared to the smartphone. To evaluate the performance of the system under bad conditions we have used UWB radios as an independent positioning system and shown that the position accuracy is better than 0.5 m under good conditions and that dead reckoning improves the accuracy significantly for short durations during bad conditions.

As more and more vehicles and other road users become connected, this technology has the potential to prevent many serious and fatal accidents between e.g. cyclists and motor vehicles as collision detection systems get access to accurate position data of road users.

Acknowledgement

This research has been funded through EISIGS (grants from the Knowledge Foundation) and through the Swedish Trafikverkets Skyltfond.

References

[1] A. Niska and J. Eriksson. *Statistik över cyklisters olyckor - Faktaunderlag till gemensam strategi för säker cykling. VTI rapport 801.* Rapport. VTI, 2013.

[2] M. Skoglund, T. Petig, B. Vedder, H. Eriksson, and E. M. Schiller. ”Static and dynamic performance evaluation of low-cost RTK GPS receivers”. In: *IEEE Intelligent Vehicles Symposium (IV).* June 2016, pp. 16–19.

[3] T. Krejci and M. Mandlik. ”Close vehicle warning for bicyclists based on FMCW radar”. In: *27th International Conference Radioelektronika (RADIOELEKTRONIKA).* Apr. 2017, pp. 1–5.

[4] W. Jeon and R. Rajamani. ”A novel collision avoidance system for bicycles”. In: *2016 American Control Conference (ACC).* July 2016, pp. 3474–3479.

[5] M. Liebner, F. Klanner, and C. Stiller. ”Active safety for vulnerable road users based on smartphone position data”. In: *IEEE Intelligent Vehicles Symposium (IV).* June 2013, pp. 256–261.

[6] H. Oda, S. Kubota, and Y. Okamoto. ”Research on Technology for Reducing Sudden Pedestrian or Cyclist Accidents with Vehicles”. In: *IEEE Intelligent Transportation Systems Conference.* Sept. 2007, pp. 1032–1036.

[7] M. Liebner, M. Baumann, F. Klanner, and C. Stiller. ”Driver intent inference at urban intersections using the intelligent driver model”. In: *IEEE Intelligent Vehicles Symposium.* June 2012, pp. 1162–1167.

[8] M. Jutila, J. Scholliers, M. Valta, and K. Kujanpää. ”ITS-G5 performance improvement and evaluation for vulnerable road user safety services”. In: *IET Intelligent Transport Systems* 11.3 (2017), pp. 126–133.

[9] Trimble. *GPS The First Global Navigation Satellite System*. Trimble Navigation Limited, 2007.

[10] Ublox. *Achieving Centimeter Level Performance with Low Cost Antennas. UBX-16010559*. White Paper. Ublox, 2016.

[11] T. Takasu and A. Yasuda. ”Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB”. In: *International Symposium on GPS/GNSS*. International Convention Centre Jeju, Korea. 2009, pp. 4–6.

[12] M. Haklay and P. Weber. ”OpenStreetMap: User-Generated Street Maps”. In: *IEEE Pervasive Computing* 7.4 (Oct. 2008), pp. 12–18.

[13] J. D., C. Gerdtman, and M. Linden. ”Signal processing algorithms for temperauture drift in a MEMS-gyro-based head mouse”. In: *International Conference on Systems, Signals and Image Processing (IWSSIP)*. May 2014, pp. 123–126.

[14] A. Vitali. *Ellipsoid or sphere fitting for sensor calibration. DT0059*. Design Tip. ST Microelectronics, 2016.

[15] Decawave. *DWM1000 IEEE 802.15.4 UWB Transceiver Module. DWM1000 Datasheet*. Datasheet. Decawave, 2016.

[16] Decawave. *The implementation of two-way ranging with the DW1000. APS013*. Application Note. Decawave, 2015.

[17] Decawave. *Sources of Error in DW1000 based Two-Way Ranging (TWR) Schemes. APS011*. Application Note. Decawave, 2015.

Paper V

A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving

Benjamin Vedder, Jonny Vinter and Magnus Jonsson

Accepted for Publication in Hindawi Journal of Robotics, 2018

Abstract

Accurate positioning is a requirement for many applications, including safety-critical autonomous vehicles. To reduce cost and at the same time improving accuracy for positioning of autonomous vehicles, new methods, tools and research platforms are needed. We have created a low-cost testbed consisting of electronics and software, that can be fitted on model vehicles allowing them to follow trajectories autonomously with a position accuracy of around 3 cm outdoors. The position of the vehicles is derived from sensor fusion between Real-Time Kinematic Satellite Navigation (RTK-SN), odometry and inertial measurement, and performs well within a 10 km radius from a base station. Trajectories to be followed can be edited with a custom GUI, where also several model vehicles can be controlled and visualized in real time. All software and Printed Circuit Boards (PCBs) for our testbed are available as open source to make customization and development possible. Our testbed can be used for research within autonomous driving, for carrying test equipment, and other applications where low cost and accurate positioning and navigation is required.

1 Introduction

It is common to use model cars for automotive research, and several studies are published in that field. For example, research within vehicle platooning has been carried out on model cars equipped with floor marking and distance sensors [1, 2], with the goal of developing Model Predictive Control (MPC) algorithms for controlling the distance between adjacent vehicles. Model cars have also been used to develop obstacle avoidance algorithms for mobile robots [3] and in student projects to teach them about autonomous driving [4]. These projects are specific and aimed at certain tasks, and the hardware and software is not available to replicate for use within other areas.

The need for a generic model vehicle platform for education and research within autonomous driving is recognized in the community, and several attempts at answering that need have been made. A project named MOPED [5] provides a model car that has three Raspberry Pi single board computers [6] connected over ethernet to simulate part of the complexity of a modern full scale car. Two of the computers on the MOPED model car run AUTOSAR [7] while the third one runs the default Raspbian Linux distribution[1] that comes with the Raspberry Pi. The reason for using AUTOSAR is to represent a software stack similar to the one on a full scale car, however it requires software tools that are not available as open source or freeware for developing the AUTOSAR portions of the software. Further, the MOPED model car only provides low level control functions for the motor and steering servo, meaning that the users have to implement trajectory following and positioning algorithms themselves, as well as equipping the car with the necessary sensors. Gulliver [8] is another initiative that addresses the need of a miniature vehicle platform in research and education environments. In their publication, the authors describe high level algorithms for handling different traffic scenarios that can be used given access

[1]https://www.raspberrypi.org/downloads/raspbian/

to a model vehicle that can follow trajectories. While these projects are an aid in education and research about autonomous driving, a significant amount of work is still required from the researchers or students to get a self-driving model car up and running. Note that with self-driving in this context, we refer to the ability to follow a predefined trajectory accurately and repeatedly, which has been one of the goals in our work as explained below.

Our work focuses on providing a hands-on hardware and software testbed that can be used to build self-driving model vehicles with minimal effort. The goal is to provide the possibility to get a model vehicle up and running, and follow a custom trajectory autonomously with 5 cm or better accuracy in just one day of work, given that the user has a background within electronics. To achieve this, we have developed low-cost hardware and both embedded and desktop software controlling model vehicles with Ackerman steering that can be modeled with a bicycle model [9]. Our testbed also has support for Hardware-In-the-Loop (HIL) testing [10, 11] by simulating parts of the vehicle dynamics, which is useful during development and automatic testing. Thus, the contribution of this work is to answer to what extent the aforementioned goal can be achieved with low-cost hardware, provide help for other researchers who want to implement their own self-driving model car, and to answer what performance and accuracy can be expected. All software and hardware design for our testbed is available on github[1] for making it possible to study and extend our platform. To our knowledge there is nothing available today that can fulfill the goal that we have for our tested, and as expressed in the aforementioned studies [5, 8] there is a need for that in the education and research communities.

In addition to usage as a research and education platform, our testbed can be used in measurement and data collection applications. For example, we have used it to pull a trailer with different radar units to be characterized around a variety of radar targets. This way the radars can be moved along a predefined trajectory around the targets, while the measurements they take are logged together with accurate position stamps. Another possible application that we have been considering is using model cars based on our testbed to carry light sensors and map the light intensity of artificial lighting in outdoor environments. The open source nature of the software in our testbed and the versatile visualization tools that are part of it make this a relatively simple task.

The remainder of this paper is organized as follows: In Section 2 we describe the architecture of our testbed and in Section 3 we describe how positioning is performed. Section 4 describes our trajectory following approach and in Section 5 we present our conclusions from this work.

2 Architecture Overview

Our testbed consists of a control Printed Circuit Board (PCB) we have developed that can be connected to a VESC[2] open-source motor controller over Controller Area Network (CAN)-bus. Together with a battery and a Global Navigation

[1]https://github.com/vedderb/rise_sdvp
[2]https://vesc-project.com

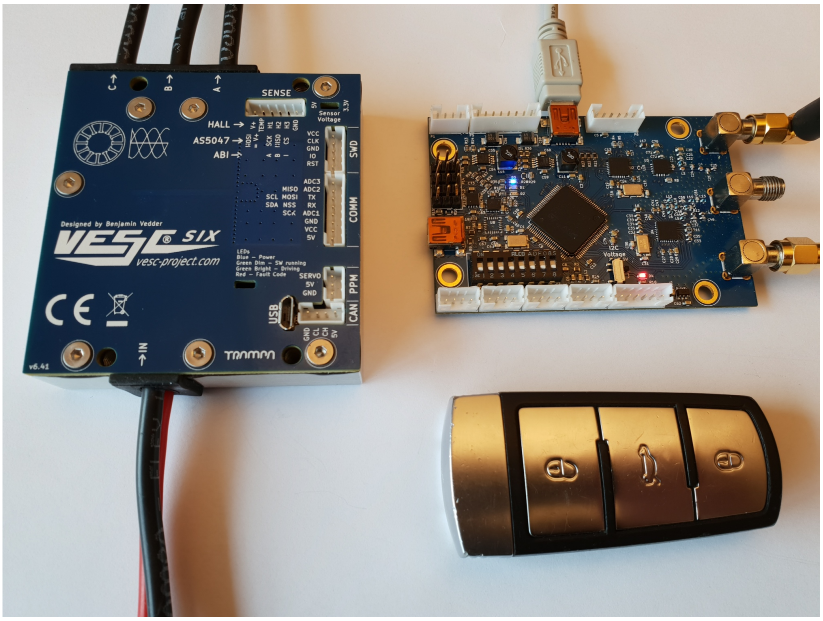


Figure 1: Our custom control PCB and our VESC 5 kW motor controller next to a car key for size comparison.

Satellite System (GNSS) antenna they can be connected to the Permanent Magnet Synchronous Motor (PMSM) and steering servo suitable for a model car. The control PCB together with a 5 kW VESC motor controller is shown in Fig. 1 together with a car key for size comparison. Another unit of the same control PCB configured as a communication interface can be connected to a laptop computer running RControlStation, which is the monitoring and control Graphical User Interface (GUI) for our testbed. The stationary control PCB can also be connected to a GNSS antenna and act as a Real-Time Kinematic Satellite Navigation (RTK-SN) base station for the testbed, eliminating the need for an external base station. This gives a minimal stand-alone configuration of our testbed, which is able to follow trajectories autonomously. The cost of this configuration, excluding the laptop cost, is in the range of 900 to 2000 depending on the choice of model car, battery size, VESC version and GNSS antennas. The schematics and hardware layout of our control PCB, the control PCB firmware and the RControlStation software, and the VESC firmware and configuration software, are all available on github[1].

While the aforementioned minimal configuration is sufficient for getting everything running and carrying out experiments, a Raspberry Pi[2] single board computer can be added for remote debugging, for video streaming, and for providing WiFi or 4G cellular connectivity. Our github repository also contains a command line utility for the Raspberry Pi that among other things provides

[1]https://github.com/vedderb
[2]https://www.raspberrypi.org/

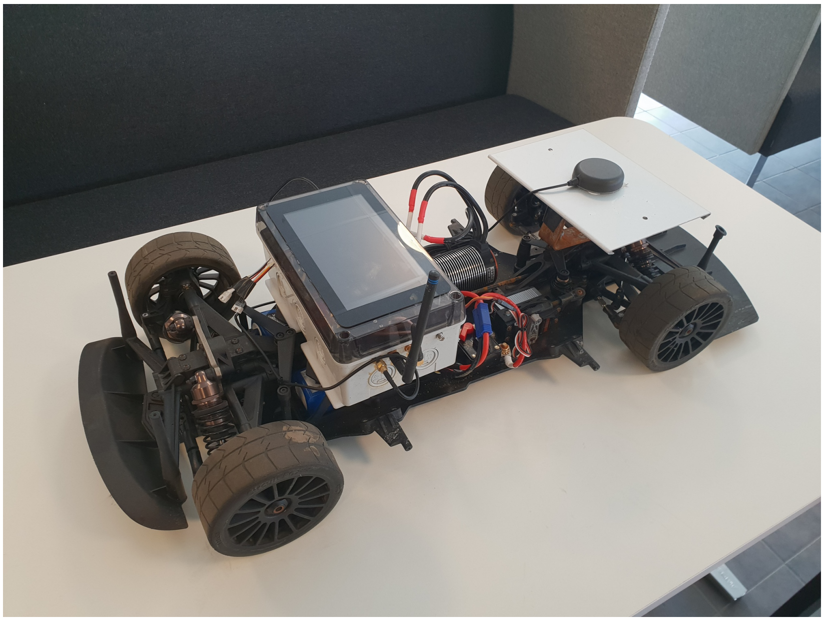


Figure 2: The 1:6 scale model car in our testbed.

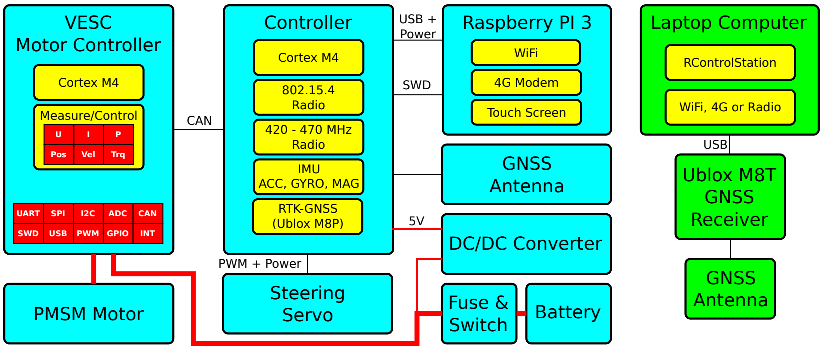


Figure 3: A block diagram of the configuration of our model car. A laptop computer for control and monitoring while acting as a RTK-SN base station is also shown.

a TCP/UDP to USB bridge for communication with RControlStation over WiFi or 4G. Fig. 1 shows a photo of one of our model cars and Fig. 3 shows a block diagram of its configuration. Note that it also has a touch screen for the Raspberry Pi for showing network connections and other useful information, such as the battery charge level. Going through the block diagram in Figure 3, our model car, shown in Figure 1, has the following components:

- A lithium-ion battery with 10 cells in series and 3 Ah, providing up to 6 hours of power depending on the driving speed.
- An integrated fuse and power switch, between the battery and the rest of the circuit.
- A DC/DC converter, that provides a 5V rail.
- Our custom controller PCB, powered from the 5V rail.
- Our VESC motor controller, powered from the battery and connected to the controller over CAN-bus.
- A PMSM motor, connected to the VESC motor controller.
- A steering servo and a GNSS antenna, connected to the controller.
- A Raspberry Pi single board computer, connected to the controller over USB. It is powered backwards through the USB ports from the controller. The Serial Wire Debug (SWD) port of the controller is also connected to the Raspberry Pi, so that the controller can be programmed and debugged remotely.
- Outside the car, there is a laptop computer connected over TCP to the Raspberry Pi using the WiFi or 4G cellular connection. Our RControlStation software runs on the laptop computer, and utilizes the connection to control and monitor the model car.
- The laptop is connected to a Ublox M8T RTK-SN receiver with an antenna mounted on a tripod to act as a base station for the model car, enabling high precision positioning (See Section 3 for more details). RControlStation handles the setup of the Ublox M8T, as well as forwarding of the required correction data for RTK-SN to the model car.

The control PCB is based on an ARM Cortex M4 microcontroller and runs the ChibiOS[1] real-time operating system. The microcontroller carries out sensor fusion for position estimation (See Section 3), the trajectory following algorithm (See Section 4) and all other functionality of the model vehicle, meaning that the connection to the laptop is only required for monitoring and sending high level control commands. In addition to the microcontroller, the control PCB containts an Inertial Measurement Unit (IMU), a CAN transceiver, two radios,

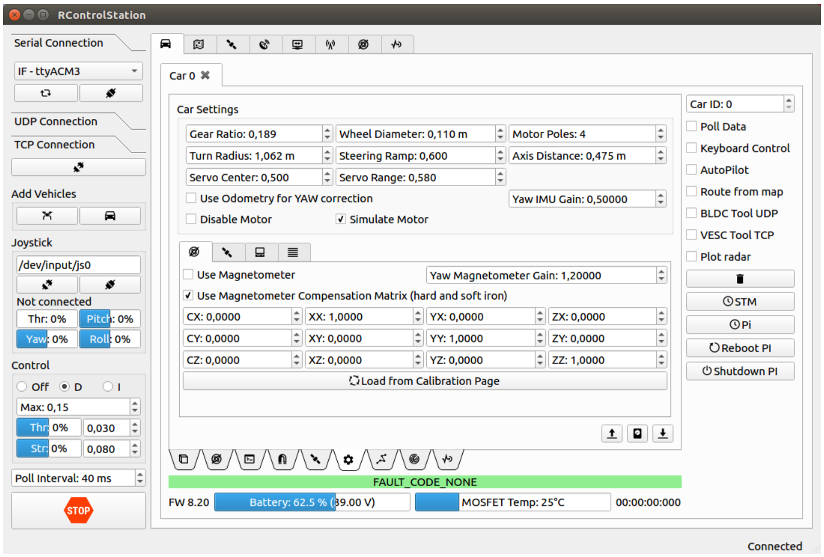
[1]http://www.chibios.org

DC/DC converters, an Ublox M8P GNSS receiver and various connection ports for possible extensions.

After the control PCB, the other essential part of the electronics and software on the model car is the VESC motor controller. The VESC is also developed by us, partly in parallel with our testbed, but this paper only describes it briefly. It can drive the motor of the model car with high efficiency over a wide dynamic range using the state-of-the-art motor control technique Field Oriented Control (FOC) [12] with Space Vector Modulation (SVM) [13]. The VESC is able to operate from zero speed with high torque without position sensors on the motor by taking advantage of a nonlinear observer [14] and effective startup algorithms. Further, it provides position and speed feedback from the motor over CAN-bus as well as closed loop speed control, which is essential for the positioning system of the model car to work accurately as described in Section 3. Another essential functionality for our testbed that the VESC provides is automatic identification of all motor parameters necessary for sensorless FOC, which are rarely available in the datasheet of inexpensive model car motors. All the configuration and parameter detection of the VESC can be performed with the accompanied VESCTool GUI, which among other things provides configuration wizards to get all settings right.

In the same way as the VESC motor controller can be configured to run with almost any PMSM without writing code and/or using expensive lab equipment for motor parameter identification, the control software can easily be configured for different model car configurations. Fig. 4 shows a screenshot of RControlStation where several of the model car parameters can be edited and stored in the control PCB, such as the wheel diameter, gear ratio and turning radius. This enables users of our testbed to configure the hardware and software to work with any model vehicle size and configuration from easy-to-use GUIs, which is one of the main objectives of our testbed. RControlStation also provides many debugging and plotting tools aiding with setting up model vehicles and performing experiments, such as magnetometer calibration and data visualization. For example, all graphs and map plots in this paper are generated in vector format using RControlStation, without requiring any additional code.

To develop and evaluate the trajectory following algorithm, as well as aiding with development of automatic test case generation with HIL, the control PCB supports a simulation mode where tests can be performed with just a USB connection to it without using the model car. This mode is implemented by simulating the behavior of the motor and mechanics with inertia and drag, and by updating the position of the vehicle with only dead reckoning from the simulated motor feedback, and heading calculated from the commanded steering angle (See Section 3 for details about the position estimation). The rate of movement of the commanded steering angle is limited to capture the behavior of a realistic steering servo. This simplifies and speeds up testing and development, while capturing many important aspects of the real-time hardware.



IF - ttyACM3

/dev/input/js0

Not connected

Thr: 0% Pitch: 0%

Yaw: 0% Roll: 0%

○ Off [✓ ● D] ○ I

Max: 0,15

Thr: 0% 0,030

Str: 0% 0,080

Poll Interval: 40 ms

Gear Ratio: 0,189 | Wheel Diameter: 0,110 m | Motor Poles: 4

Turn Radius: 1,062 m | Steering Ramp: 0,600 | Axis Distance: 0,475 m

Servo Center: 0,500 | Servo Range: 0,580

☐ Use Odometry for YAW correction | Yaw IMU Gain: 0,50000

☐ Disable Motor | ☑ Simulate Motor

☐ Use Magnetometer | Yaw Magnetometer Gain: 1,20000

☑ Use Magnetometer Compensation Matrix (hard and soft iron)

CX: 0,0000 | XX: 1,0000 | YX: 0,0000 | ZX: 0,0000

CY: 0,0000 | XY: 0,0000 | YY: 1,0000 | ZY: 0,0000

CZ: 0,0000 | XZ: 0,0000 | YZ: 0,0000 | ZZ: 1,0000

Car ID: 0

☐ Poll Data

☐ Keyboard Control

☐ AutoPilot

☐ Route from map

☐ BLDC Tool UDP

☐ VESC Tool TCP

☐ Plot radar

FAULT_CODE_NONE

FW 8.20 | Battery: 62.5 % (39.00 V) | MOSFET Temp: 25°C | 00:00:00:000

Figure 4: A screenshot of RControlStation, where parameters for the model car can be edited.

3 Positioning

Accurate positioning is an essential part of our testbed. Our main position source is RTK-SN, which is based on consumer GNSS technology with the addition of carrier-phase measurements of the satellite signals. The carrier-phase measurements together with conventional code measurements on both the rover (the object to be positioned) and a base station with a known position within a 10 km radius from the rover are required. This means that a data link between the rover and base station with a data rate of around 100 bytes/second, depending on the number of satellites, is required. The data stream with code and carrier-phase measurements from the base station, together with information about the position of the base station, is usually referred to as correction data and sent using a format such as RTCM3[1]. RControlStation can either act as a base station for the model vehicles by connecting an appropriate GNSS receiver to it (such as the Ublox M8T), or by connecting to an existing base station using the TCP or NTRIP[2] protocol. The correction data is then sent from RControlStation to the vehicles using the communication link of choice (4G, WiFi or radio) using the RTCM3 format. The position accuracy of the rover relative to the base station with RTK-SN is around 1 cm under optimal conditions.

Traditionally RTK-SN has been an expensive technology, but recently less expensive solutions have become available [15]. In previous work we have

[1]http://www.navipedia.net/index.php/RTK_Standards

[2]https://en.wikipedia.org/wiki/Networked_Transport_of_RTCM_via_Internet_Protocol

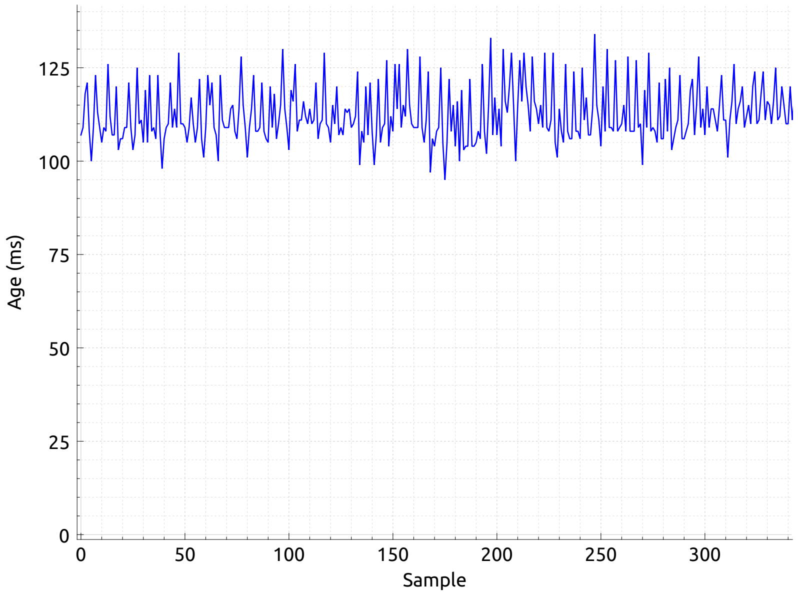


Figure 5: Delay jitter of the RTK-SN samples over time.

compared the performance of high and low cost RTK-SN systems [16] and came to the conclusion that, besides the longer initial convergence time and low update rates of the low cost systems, the performance is similar. We have also studied how low cost RTK-SN performs in urban environments [17] using the same control PCB as presented here, and came to the conclusion that it performs well even when the view of the sky is poor given that multiple satellite constellations and/or sensor fusion with dead reckoning is used.

3.1 Challenges

The main challenge with using inexpensive RTK-SN equipment with our testbed is the low position update rate, as well as latency and jitter between updates. The Ublox M8P RTK-SN receiver provides a position update rate of 5 Hz, which is too low for accurate positioning and control at speeds of up to 80 km/h, which our model cars are capable of. Fig. 5 shows the measured age of consecutive position updates from the Ublox M8P. As can be seen, the samples are between 95 and 135 ms old and have jitter of up to 40 ms. If the model car moves at 10 m/s, which is less than half of the maximum speed, a latency of 100 ms causes a position error of 1 m. At the same speed, the jitter between consecutive position samples can cause errors of up to 0.5 m. Having a low update rate and high latency on one or more of the sensors on the system is a common problem within robotics, especially when low cost hardware is involved [18], and there are various methods to handle that.

To deal with the low update rate, latency and jitter of the RTK-SN position samples, dead reckoning from sensor fusion between the odometry data from the VESC motor controller and samples from the IMU is combined with the RTK-SN position samples to get a total update rate of 100 Hz without latency. We have come up with the method below of performing the combination. Note that the RTK-SN position in **3)** and **4)** first is moved to the center of the vehicle based on the current estimation of the heading angle and offset of the GNSS antenna from the vehicle center.

1) The IMU is sampled at 1 kHz and fed to a quaternion-based orientation filter as proposed by Madgwick et al. [19], which provides the roll, pitch and heading of the model vehicle. When the magnetometer of the IMU is configured to be included, we have improved the position filter by adding tilt compensation for estimating the heading [20] as well as ellipsoid fitting for hard and soft iron compensation of the magnetometer [21].

2) The motor position is sampled from the VESC motor controller at 100 Hz, and combined with the estimated heading from the position filter to calculate a relative dead reckoning position. This position is stored in a 100 samples long FIFO buffer together with the current GNSS time stamp derived from the Pulse Per Second (PPS) signal from the Ublox M8P receiver.

3) When RTK-SN position samples are received, they are compared to the dead reckoning position sample from the FIFO buffer from **2)** with the closest time stamp to the sample, which will be 5 ms away in the worst case. The difference between the closest FIFO buffer sample and the RTK-SN sample is then used to correct the current position by moving it in the direction of the difference as:

$$p_{xy} = p_{xy\_old} + d_{xy}G_{stat} + d_{xy}G_{dyn}d_p \tag{1}$$

where p_{xy} is a vector with the new xy-position of the vehicle, p_{xy_old} is the previous position of the vehicle, d_{xy} is the difference vector between the latest RTK-SN sample and the closest position in time from the FIFO buffer, G_{stat} is a scalar configurable static gain, G_{dyn} is a scalar configurable dynamic gain and d_p is a scalar of how far the vehicle has moved since the previous RTK-SN update. Noise between consecutive RTK-SN samples is rejected by gradually moving the current position using this method instead of moving it the full difference at once for every sample.

4) When the magnetometer is not used to provide an absolute heading reference, the heading of the orientation filter in **1)** is updated every time a RTK-SN sample is received by first computing an RTK-SN heading as:

$$\phi_{rtk} = -atan2(y_{rtk} - y_{prtk}, x_{rtk} - x_{prtk}) \tag{2}$$

where (x_{rtk}, y_{rtk}) is the latest RTK-SN position sample and (x_{prtk}, y_{prtk}) previous RTK-SN position sample. After that a heading difference is calculated as $\phi_d = \phi_{rtk} - \phi_{FIFO}$, where ϕ_{FIFO} is the heading of the sample from the FIFO buffer from **2)** closest to the average time between the latest and previous RTK-SN samples. The heading in the current position is then updated by rotating it in the angular direction of ϕ_d with a step proportional to how far the vehicle has moved between the latest and previous RTK-SN samples and the heading correction gain. Scaling by the distance moved is used because consecutive RTK-SN samples do not provide any heading information when the vehicle is stationary.

The rationale of this approach is that the dead reckoning position is accurate over short distances, but drifts as the distance increases. The time delay of the RTK-SN samples is short enough to not cause significant degradation of the dead reckoning, meaning that the current position can be calculated accurately at a high rate by using an old absolute position and the relative movements that have occurred since then. Notice that the heading estimation is critical for the dead reckoning to perform well, more details about that can be found in our previous work [17].

3.2 Performance Evaluation

To evaluate the performance of the latency and jitter compensation described above, we have driven the model car along a trajectory with rapid accelerations and decelerations, and measured the difference between the RTK-SN position and the dead reckoning position for each sample with and without the FIFO delay compensation. Fig. 6 shows the difference without compensation, and Fig. 7 shows the difference with compensation. As can be seen, without the compensation the difference is stable at constant low speed, but goes to 1.1 m during the accelerations and decelerations. With the compensation enabled, the difference is limited to 10 cm during the acceleration and to 20 cm during the deceleration, due to wheel slippage. The reason that the difference in Fig. 6 is low during constant speed without compensation is that the position converges to an invalid value, which will be apparent when the speed changes rapidly; whereas when the compensation is enabled the position will be closer to the true position at all times. It can also be noted that during the constant speed shown in Fig. 6 there is over 10 cm position jitter while Fig. 7 shows less than 3 cm jitter; this is due to the delay jitter between the RTK-SN samples shown in Fig. 5.

To obtain an estimate about the absolute accuracy and repeatability of the position, we have downloaded a 20 m radius circle trajectory to the model car with a constant speed of 4 km/h. While the car was following that trajectory lap after lap on artificial grass, it made visible traces that allowed us to visually inspect the deviation of the car tires from the traces, as seen in Fig. 8. As far as we could observe, the tires of the car stayed within the traces with less than half the tire width, or 2.5 cm, for the entire experiment. We also measured the diameter of the circle on the grass, and it had the correct diameter as accurately as we were able to measure with our equipment. The position difference between the RTK-SN samples and the closest samples in the dead reckoning FIFO stayed below 3 cm for the entire experiment, giving us confidence that our model vehicle can estimate its absolute position with 3 cm accuracy.

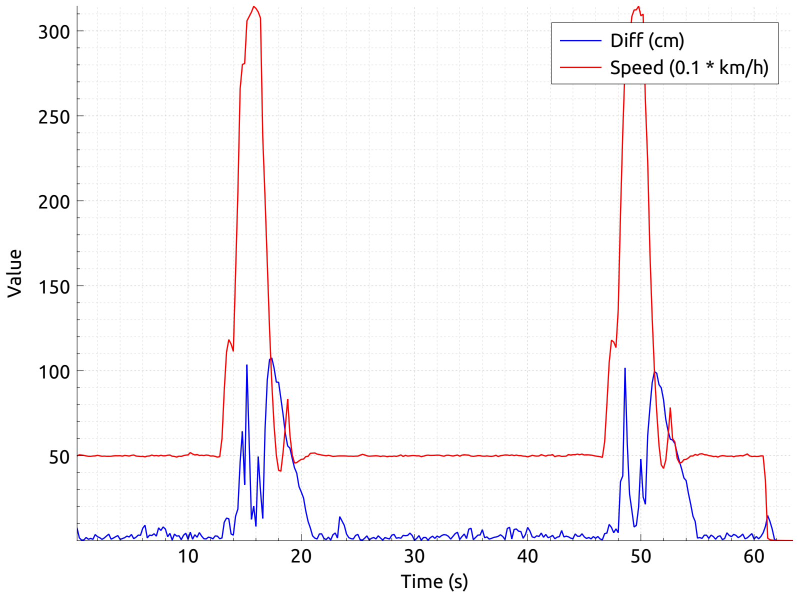


Figure 6: The speed and difference between the RTK-SN and dead reckoning position during hard acceleration and deceleration. Delay compensation for the RTK-SN samples is disabled.

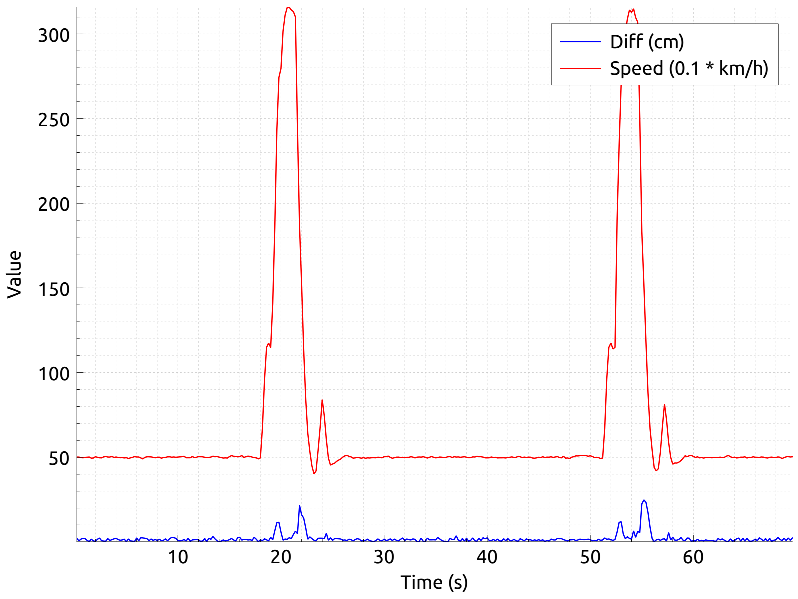


Figure 7: The speed and difference between the RTK-SN and dead reckoning position during hard acceleration and deceleration. Delay compensation for the RTK-SN samples is enabled.

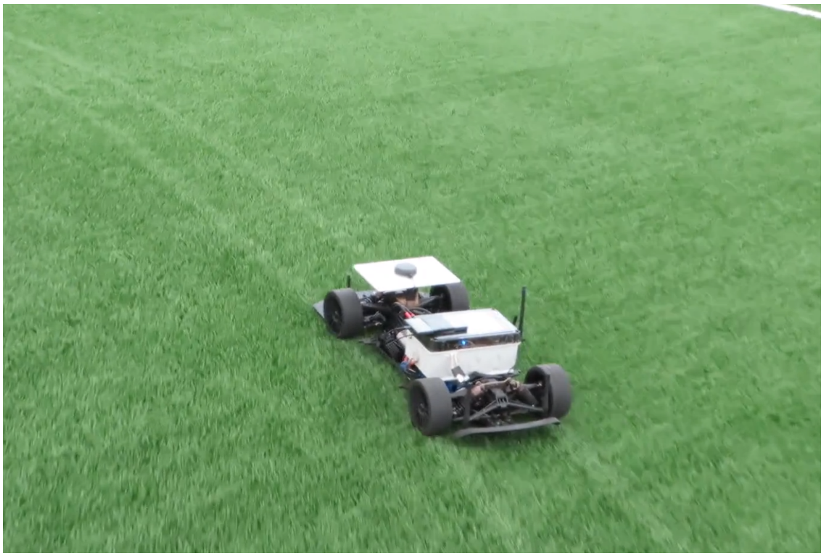


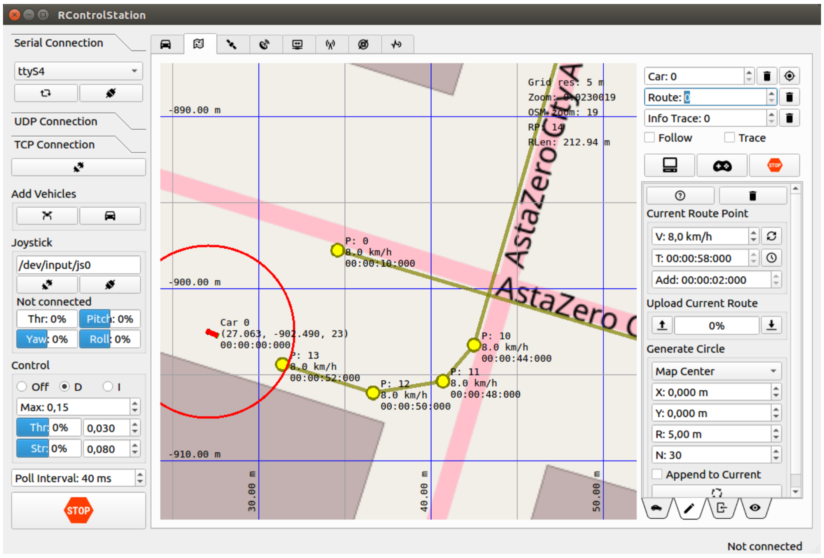
Figure 8: Our 1:6 model car repeatedly following a circular trajectory on artificial grass. The traces from previous laps are visible, and can be used to estimate the lateral positioning and control repeatability.

4 Trajectory Following

An important aspect of our testbed is the ability to edit and follow trajectories. We define a trajectory as a list of points where each point has a xy-positions and a speed or time stamp, depending on the mode of operation. Our testbed support three trajectory following modes: **1) Speed based**, where the model vehicle adjusts its speed proportional to the set speed and relative distances to the two closest trajectory points; **2) Absolute time**, where the model vehicle adjusts its speed such that it reaches the trajactory points at absolute UTC[1] times (derived from the GNSS receiver clock); and **3) Relative time**, where the model vehicle adjusts its speed such that it reaches the trajectory points at times relative to when the start command was issued. Creating trajectories is most intuitive using the speed mode, but the time-based modes are necessary to synchronize multiple vehicles in a scenario. There is also a synchronization command available that can be sent to the model vehicles in real time, so that they continuously update their speed to reach a trajectory point specified in the command at the time specified in the command, based on the distance left along the trajectory to that point.

RControlStation allows users to graphically edit trajectories with an overlay of OpenStreetMap [22], as shown in the screenshot in Fig. 9. We chose OpenStreetMap because it is accompanied by a complete set of tools for map creation and rendering, available as open source software. This makes it easy to update and create artificial maps in e.g. test areas, and render them on a server.

[1]https://en.wikipedia.org/wiki/Coordinated_Universal_Time


RControlStation

Serial Connection
ttyS4
UDP Connection
TCP Connection
Add Vehicles
Joystick
/dev/input/js0
Not connected
Thr: 0% Pitch: 0%
Yaw: 0% Roll: 0%
Control
Off [✓ D] I
Max: 0,15
Thr 0% 0,030
Str 0% 0,080
Poll Interval: 40 ms

Car: 0
Route: 0
Info Trace: 0
Follow Trace
Current Route Point
V: 8,0 km/h
T: 00:00:58:000
Add: 00:00:02:000
Upload Current Route
0%
Generate Circle
Map Center
X: 0,000 m
Y: 0,000 m
R: 5,00 m
N: 30
Append to Current

Not connected

Figure 9: A screenshot of the map page in RControlStation, where trajectories can be edited.

The transforms for converting OpenStreetMap rendered map tiles with Web Mercator[1] projection to the coordinate system of our testbed for viewing in RControlStation are well documented on their wiki page[2]. This was a significant aid in implementing the map rending functionality of RControlStation.

4.1 Lateral Control

Lateral control along the trajectories of the model vehicles in our testbed is performed using the pure pursuit algorithm, which is a common method within robotics [23]. Essentially it works in the following way **1)** draw a circle around the vehicle with a radius of the chosen look-ahead distance; **2)** calculate the point where that circle intersects the trajectory; if multiple points are found pick the one furthest ahead on the trajectory; and **3)** adjust the steering angle of the vehicle such that it follows an arc that intersects with that point. When visualized, it looks like the vehicle follows a point that moves away from it along the trajectory. More details about the pure pursuit algorithm can be found in the literature [23]. We have also added support for the case when the vehicle is further away from the trajectory than the look ahead distance, in which case it will follow the closest point on the trajectory until the circle around the vehicle intersects with the trajectory, after which the algorithm is carried out as usual. This is useful when sharp turns or oscillations cause the vehicle to lose the trajectory, or when it is started far away from the trajectory. Also, if the model vehicle is to be used with automatic test case generation, it is helpful to have

[1]https://en.wikipedia.org/wiki/Web_Mercator
[2]https://wiki.openstreetmap.org/wiki/Main_Page

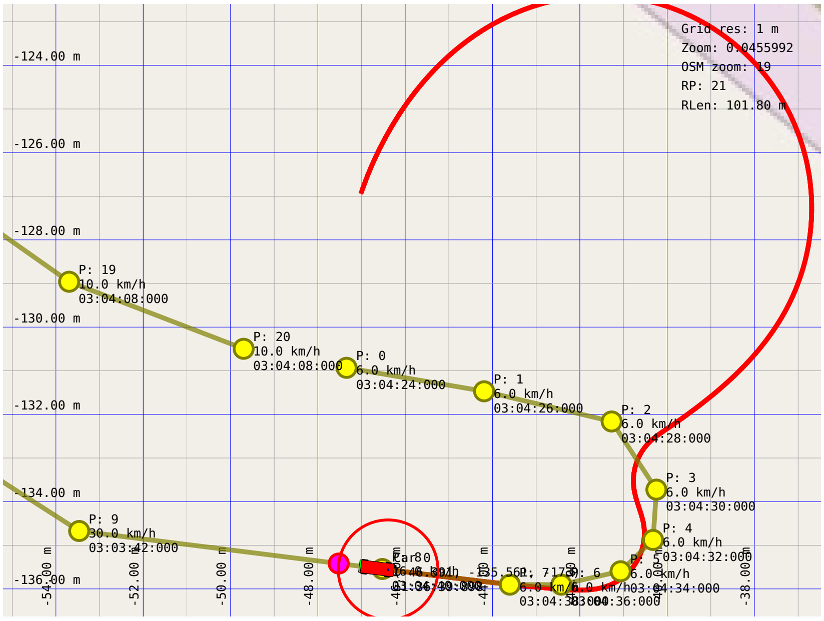


Figure 10: The red trace shows how the model car navigates back to the trajectory with angle distance gain disabled.

the ability to drive to the closest point on a recovery trajectory and drive back to the initial position along it.

Improvements to the pure pursuit algorithm commonly found in literature are interpolation of the trajectory to find points between trajectory points [23] and to use an adaptive look-ahead distance [24]. Both of these improvements have been employed in our testbed to increase the trajectory following performance. We have also implemented one to our knowledge unique improvement to the pure pursuit algorithm, which is adding gain to the steering angle calculation when the point to be followed is far away. The steering angle is corrected as:

$$A_{corr} = \begin{cases} A_{st}(1+0.2D) & \text{if } D < 20\,\text{m} \\ 5A_{st} & \text{otherwise} \end{cases} \tag{3}$$

where A_{corr} is the corrected steering angle, A_{st} is the steering angle that leads to the goal point along a circle that tangents with the car and D is the distance to the goal point. The gain helps when the vehicle starts far away from the trajectory heading away from it, where it without the gain would follow an arc longer than necessary to reach the trajectory. Fig. 10 shows the arc the vehicle follows without the distance gain and Fig. 11 shows the arc that it follows with the gain. The starting position is the same in both figures, namely where the red line starts. As can be seen, the arc in Fig. 11 is significantly shorter. Note that the look-ahead distance of the vehicle in the figures is illustrated with a red circle around it and that the point it is aiming for is drawn on the trajectory.

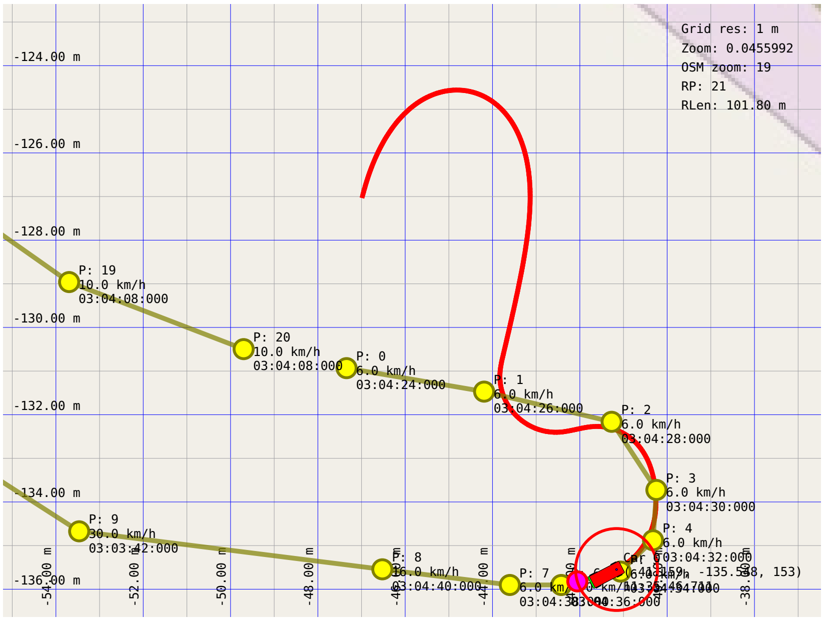


Figure 11: The red trace shows how the model car navigates back to the trajectory with angle distance gain enabled.

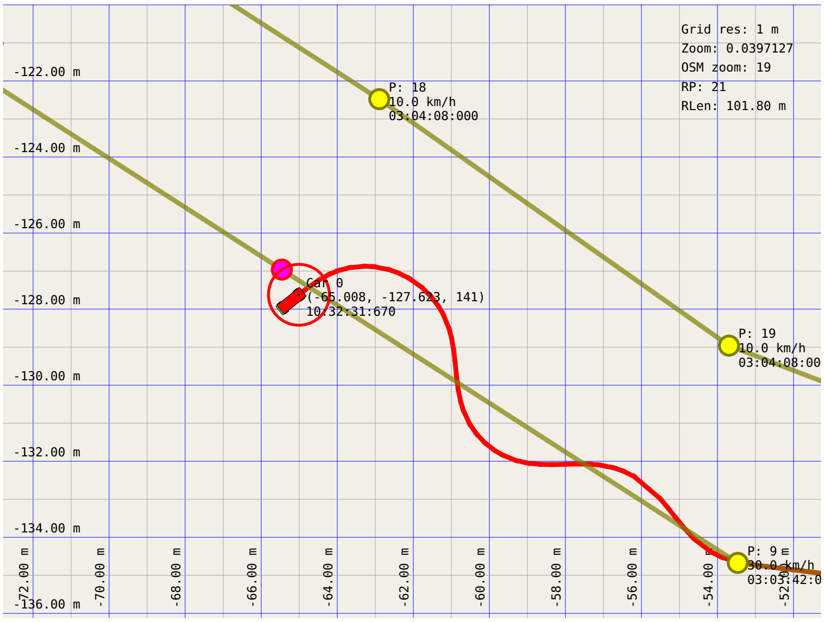


Figure 12: The red trace shows how the model car oscillates at 30 km/h with too small fixed look-ahead distance.

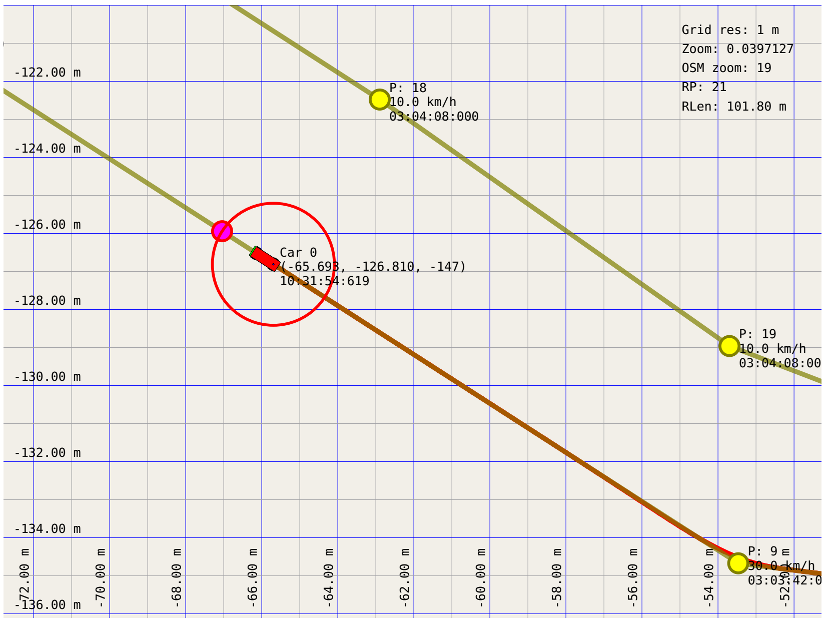


Figure 13: The red trace shows how the model car follows the trajectory at 30 km/h with sufficient look-ahead distance.

Using the simulation mode of the control PCB and a trajectory that has variable speeds and sharp turns, we have set up an experiment to evaluate the performance of the lateral control with different settings and improvement strategies of the pure pursuit algorithm. First we disabled adaptive look-ahead distance and found a static distance that is long enough to follow the trajectory in a stable manner. As an unstable example, Fig. 12 shows how the car behaves when the look-ahead distance is too short on a high speed part of the trajectory, whereas the same scenario with a sufficient look-ahead distance is shown in Fig. 13. The maximum deviation from that trajectory without adaptive look-ahead distance during a stable lap was 40 cm, whereas it was 13 cm for the same track with adaptive look-ahead distance enabled. The difference in deviation from the trajectory comes from the low-speed parts of that trajectory with sharp turns, where a short look-ahead distance at low speed allows the vehicle to follow the trajectory tightly while still being stable due to the low speed. Fig. 14 shows a tight turn without adaptive look-ahead distance and Fig. 15 shows the same turn with adaptive look-ahead distance. For reference, the adaptive look-ahead distance is calculated based on the speed of the vehicle as:

$$d = d_{base}(1 + |v| * 0.05)^2 \tag{4}$$

where d is the calculated adaptive look-ahead distance, v is the speed in m/s and d_{base} is the look-ahead distance when the speed is 0. This equation was derived experimentally.

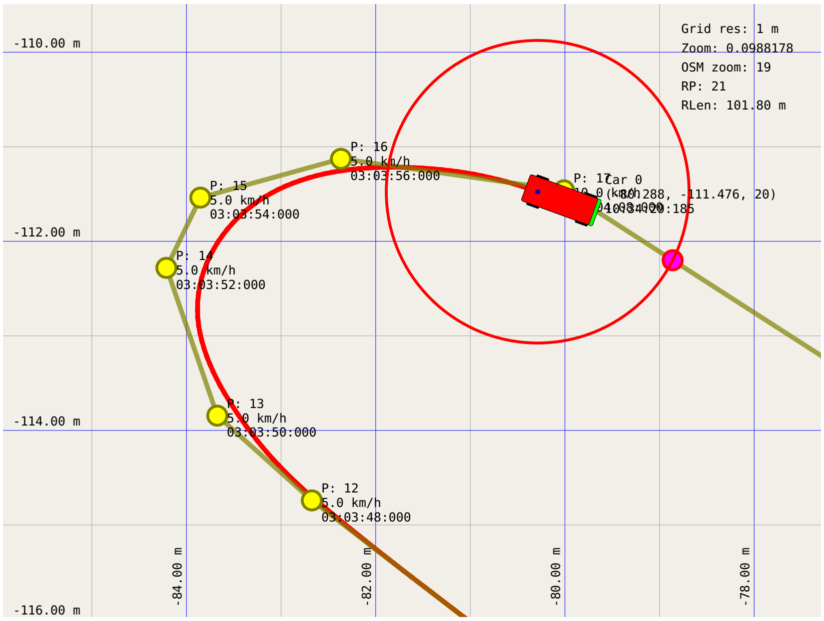


Figure 14: The red trace shows how the model car followed the trajectory with an arc without adaptive look-ahead distance enabled.

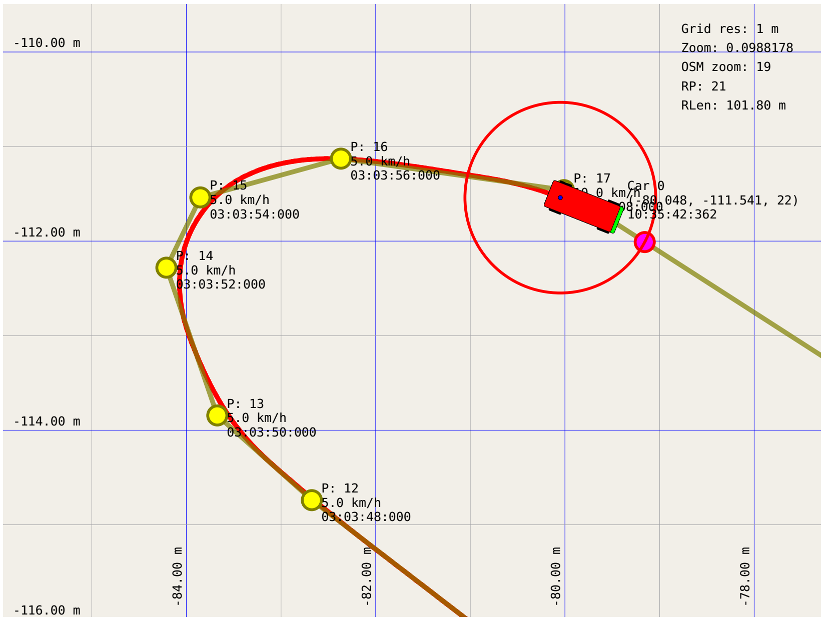


Figure 15: The red trace shows how the model car followed the trajectory with an arc with adaptive look-ahead distance enabled.

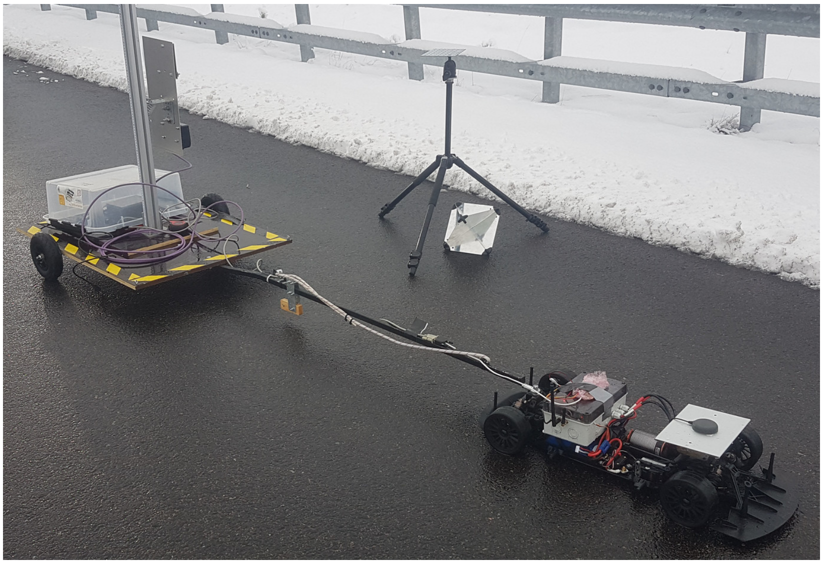


Figure 16: The model car pulling a trailer with a radar to be characterized.

5 Conclusion

In this paper we have presented a novel testbed for research and development within the areas of autonomous driving and accurate positioning. We were able to achieve high position accuracy and low latency using low-cost hardware by fusing data from multiple sensors (Accelerometer, Gyroscope, Odometry, RTK-SN) to take advantages of their individual strengths. For trajectory following we have implemented the well-known pure pursuit algorithm along with two common and one unique improvements as described in Section 4.

Model vehicles based on our testbed can follow trajectories autonomously, and all tools necessary for visualization and trajectory generation are provided. Our testbed is scalable to a wide range of model vehicles, and can be set up in just one day of work given familiarity with the testbed. The custom hardware and all involved software is open source for easy development and extension. We have also performed a wide range of tests to ensure high performance and reliability in various situations. To our knowledge there is no similar testbed available today, and there is a significant need for it in the research and education communities.

In addition to use within research and education, our testbed can be used in data acquisition applications where sensors need to be moved accurately according to a defined path, while data is stored together with position and speed. For example, Fig. 16 shows a photo of our model car pulling a trailer with a radar to be characterized along a predefined path around different targets. The open source nature of our testbed provides a solid base for such applications, and enables further functionality to be implemented without the burden of implementing the positioning and navigation functionality.

Acknowledgement

This research has been funded through EISIGS (grants from the Knowledge Foundation) and by VINNOVA via the FFI projects Chronos step 1 and 2.

Data Availability

As our testbed is open source, including all PCB designs, embedded software and desktop software, the experiments presented in this paper can be replicated by constructing a model car using the resources available at https://github.com/vedderb/rise_sdvp. The data, plots and trajectories, as well as additional data and plots from the experiments presented in this paper can be found here https://github.com/vedderb/rise_sdvp/tree/master/Misc/Test%20Data/paper_2018-06. To replicate the experiments and collect similar data, as well as to generate similar plots, the trajectories in the *routes* directory from the previous link can be used together with the appropriate terminal commands from the car terminal of RControlStation. RControlStation is part of the open source software, and all terminal commands can be listed by typing help in the car terminal.

Conflict of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

References

[1] F. C. Braescu and C. F. Caruntu. ”Prototype model car design for vehicle platooning”. In: *International Conference on Optimization of Electrical and Electronic Equipment (OPTIM) 2017 Intl Aegean Conference on Electrical Machines and Power Electronics (ACEMP).* May 2017, pp. 953–958.

[2] F. C. Braescu. ”Basic control algorithms for vehicle platooning prototype model car”. In: *21st International Conference on System Theory, Control and Computing (ICSTCC).* Oct. 2017, pp. 180–185.

[3] A. Fenesan, T. Pana, D. Szöcs, and W. H. Chen. ”Building an electric model vehicle and implementing an obstacle avoidance algorithm”. In: *International Conference and Exposition on Electrical and Power Engineering.* Oct. 2012, pp. 49–53.

[4] F. Bormann, E. Braune, and M. Spitzner. ”The C2000 autonomous model car”. In: *4th European Education and Research Conference (EDERC 2010).* Dec. 2010, pp. 200–204.

[5] J. Axelsson, A. Kobetski, Z. Ni, S. Zhang, and E. Johansson. ”MOPED: A Mobile Open Platform for Experimental Design of Cyber-Physical Systems”. In: *40th EUROMICRO Conference on Software Engineering and Advanced Applications.* Aug. 2014, pp. 423–430.

[6] G. Halfacree and E. Upton. *Raspberry Pi User Guide.* 1st. Wiley Publishing, 2012.

[7] R. Svenningsson, R. Johansson, T. Arts, and U. Norell. ”Formal Methods Based Acceptance Testing for AUTOSAR Exchangeability”. In: *SAE Int. Journal of Passenger Cars– Electronic and Electrical Systems* 5.2 (2012).

[8] M. Pahlavan, M. Papatriantafilou, and E. M. Schiller. ”Gulliver: A Test-Bed for Developing, Demonstrating and Prototyping Vehicular Systems”. In: *IEEE 75th Vehicular Technology Conference (VTC Spring).* May 2012, pp. 1–2.

[9] K. Hulme, E. Kasprzak, K. English, D. Moore-Russo, and K. Lewis. ”Experiential Learning in Vehicle Dynamics Education via Motion Simulation and Interactive Gaming”. In: *International Journal of Computer Games Technology* 2009 (2009).

[10] A. Soltani and F. Assadian. ”A Hardware-in-the-Loop Facility for Integrated Vehicle Dynamics Control System Design and Validation”. In: *7th IFAC Symposium on Mechatronic Systems MECHATRONICS 2016* 49.21 (2016), pp. 32–38.

[11] A. Mouzakitis, D. Copp, R. Parker, and K. Burnham. ”Hardware-in-the-Loop System for Testing Automotive Ecu Diagnostic Software”. In: *SAGE Measurement and Control Journal* 42.8 (2009), pp. 238–245.

[12] J. P. John, S. S. Kumar, and B. Jaya. ”Space Vector Modulation based Field Oriented Control scheme for Brushless DC motors”. In: *International Conference on Emerging Trends in Electrical and Computer Technology.* Mar. 2011, pp. 346–351.

[13] M. Gaballah and M. El-Bardini. ”Low cost digital signal generation for driving space vector PWM inverter”. In: *Shams Engineering Journal* 4.4 (2013), pp. 763–774.

[14] J. Lee, J. Hong, K. Nam, R. Ortega, L. Praly, and A. Astolfi. ”Sensorless Control of Surface-Mount Permanent-Magnet Synchronous Motors Based on a Nonlinear Observer”. In: *IEEE Transactions on Power Electronics* 25.2 (Feb. 2010), pp. 290–297.

[15] T. Takasu and A. Yasuda. ”Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB”. In: *International Symposium on GPS/GNSS.* International Convention Centre Jeju, Korea. 2009, pp. 4–6.

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

[17] B. Vedder, J. Vinter, and M. Jonsson. ”Accurate positioning of bicycles for improved safety”. In: *IEEE International Conference on Consumer Electronics (ICCE).* Jan. 2018, pp. 1–6.

[18] M. Bošnak, D. Matko, and S. Blažič. ”Quadrocopter Hovering Using Position-Estimation Information from Inertial Sensors and a High-delay Video System”. In: *Journal of Intelligent & Robotic Systems* 67.1 (2012), pp. 43–60.

[19] S. Madgwick, A. J. L. Harrison, and R. Vaidyanathan. ”Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm”. In: *IEEE International Conference on Rehabilitation Robotics (ICORR).* 2011, pp. 1–7.

[20] T. Ozyagcilar. *Implementing a Tilt-Compensated eCompass using Accelerometer and Magnetometer Sensors. AN4248.* Application Note. Freescale Semiconductor, 2015.

[21] A. Vitali. *Ellipsoid or sphere fitting for sensor calibration. DT0059*. Design Tip. ST Microelectronics, 2016.

[22] M. Haklay and P. Weber. ”OpenStreetMap: User-Generated Street Maps”. In: *IEEE Pervasive Computing* 7.4 (Oct. 2008), pp. 12–18.

[23] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro. ”Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas”. In: *IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. Oct. 2016, pp. 7–12.

[24] M. W. Park, S. W. Lee, and W. Y. Han. ”Development of lateral control system for autonomous vehicle based on adaptive pure pursuit algorithm”. In: *14th International Conference on Control, Automation and Systems (ICCAS 2014)*. Oct. 2014, pp. 1443–1447.

Paper VI

Automated Testing of Ultra-Wideband Positioning for Autonomous Driving

Benjamin Vedder, Joel Svensson, Jonny Vinter and Magnus Jonsson

To be submitted for review, 2018

Abstract

Autonomous vehicles need accurate and dependable positioning, and these systems need to be tested extensively. We have evaluated positioning based on Ultra-Wide Band (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 that 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, restore the state between tests and to maintain safety by preventing collisions. We were able to automatically generate and carry out hundreds of experiments on the model car in real time, and re-run 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 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 with an Ultra-Wide Band (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.

To generate the tests, we utilize the Property-Based Testing (PBT) tool ScalaCheck [4]. PBT is an approach to test functional requirements of software [5]. With PBT test cases are automatically generated from an abstract model of the System Under Test (SUT), as opposed to being manually written as is 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 [6]. FI is commonly used across the entire development process of safety-critical systems; from models of hardware [7] and models of software [8] to software deployed and running on the target system [9, 10]. 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 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 [11]. This way functional and non-functional 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 [12] 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. This brings the following challenges:

I Instead of simulated time, the system is now running in real time. 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, 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, as well as 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:

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

- Our method to generate safe and random trajectories for the model car, as well as how to generate a trajectory to drive the car back to the initial position between tests.
- Showing how to repeatedly replay experiments on real-time hardware with and without FI to study the effects of faults, as well as the effects of random variations in the test environment.
- 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 our self-driving model car equipped with an UWB ranging module, as 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 [13] with dead reckoning based on the Inertial Measurement Unit (IMU) and odometry feedback from the motor controller. 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 [14].

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] that the car can follow using a variation 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

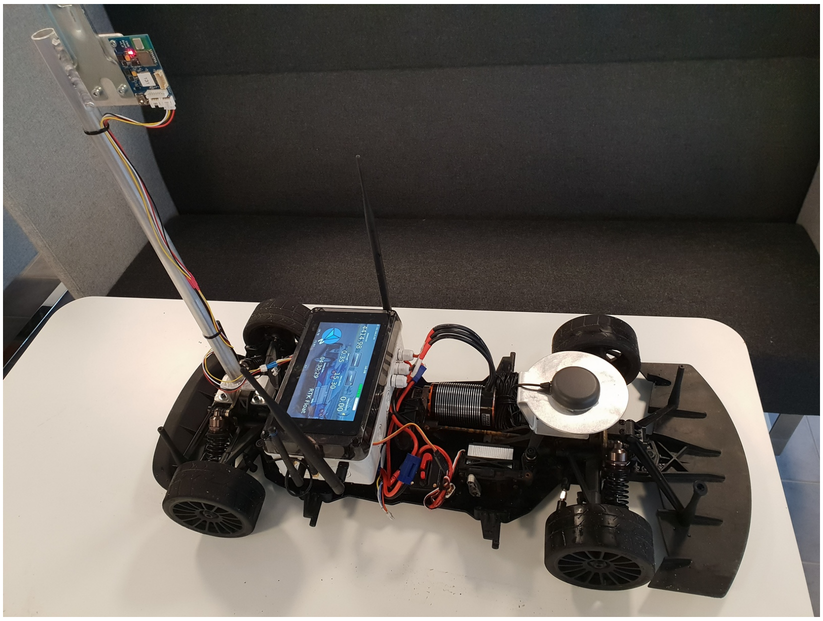


Figure 1: Photo of our self-driving model car with our UWB module attached to the back on a stick.

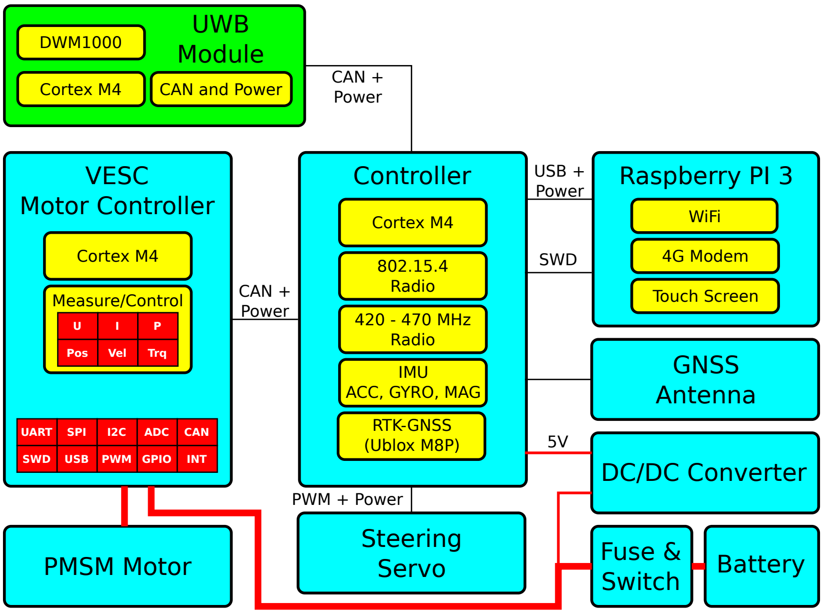


Figure 2: Block diagram of our self-driving model car with our UWB module in green.

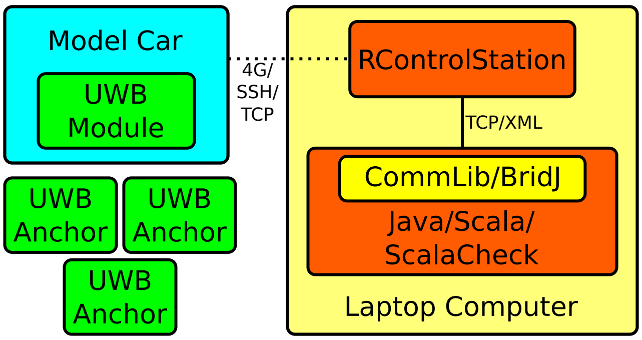


Figure 3: Experimental setup consisting of a laptop computer, our model car and two or more of our UWB anchors.

generated trajectories to the map and/or the model car. This C interface together with the BridJ[1] 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,

[1]https://github.com/nativelibs4java/BridJ

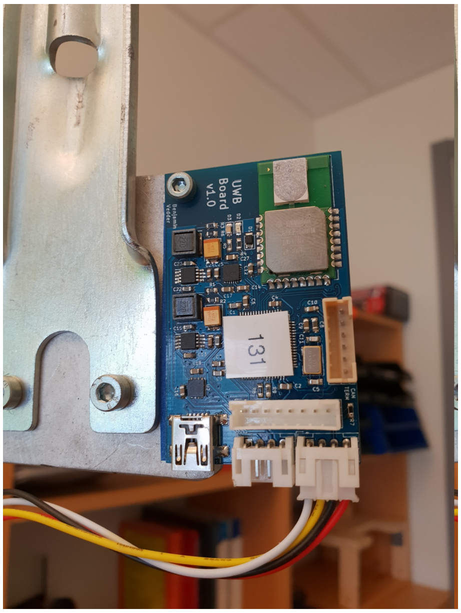


Figure 4: Our custom UWB module mounted on the model car and connected over CAN-bus. Power is also provided through the same connector.

as shown in Figure 1 and Figure 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 RControl-Station 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 that the distance measurements to the anchors are done one at a time by cycling through a list with anchors provided by RControlStation, and only making measurements to the anchors that are closer than 80 m away from the last position estimate. The 80 m cut is done

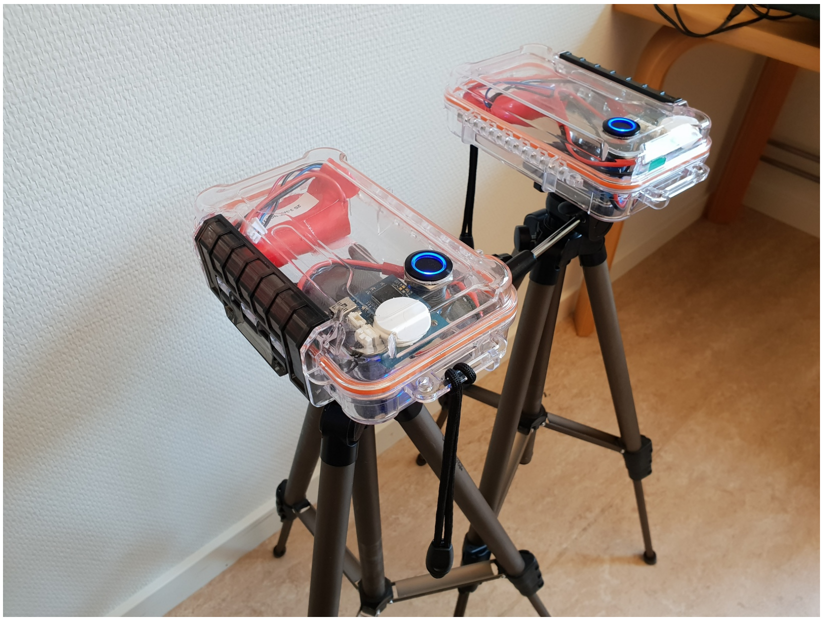


Figure 5: Our UWB modules mounted on tripods, each in a waterproof enclosure together with a battery. These are used as stationary anchors.

because the measurements to 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:

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

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, 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 to the anchor for which this measurement was made with is calculated as:

$$d_e = \|a_{xy} - p_{uwb}\| \tag{3}$$

$$v_{uwb} = (a_{xy} - p_{uwb})/d_e \tag{4}$$

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 1 pointing in the estimated direction of the anchor. Then the estimated position of the car is updated as:

$$c = \begin{cases} d_e - d_m & \text{if } |d_e - d_m| < 0.2\,\text{m} \\ 0.2\frac{|d-d_m|}{d-d_m} & \text{otherwise} \end{cases} \tag{5}$$

$$p_{xy} = p_{xy\_old} + cv_{uwb} \tag{6}$$

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 to an anchor arrives, the measurement is compared to the expected distance to 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 [4], which is a framework for testing, primarily, Java and Scala programs implemented in the object-oriented and functional programming language Scala. We go 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[1], 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

[1]org.scalacheck.commands.commands

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:

```
class Car {
  // Apply brake and wait until the car has stopped
  def stopCar(): Unit = {...}

  // Send the next trajectory part to the car and wait until
  // it is almost finished. Returns true if the maximum difference
  // between the UWB and RTK−SN position stayed below 1m
  def runSegments(route: List[RpPoint]): Boolean = {...}

  // Drive the car back to its initial position and wait until it
  // arrives. Returns true if sucessful, false otherwise.
  def runRecoveryRoute(ri: RouteInfo, carRoute: Int): Boolean = {...}

  // Add a fault to one of the probes
  def addFault(probe: String, faultType: String,
    param: Double, start: Int, duration: Int): Unit = {...}

  // Clear all faults
  def clearFaults(): Unit = {...}

  // Set the UWB position equal to the RTK−SN position
  // in order to start from a known state.
  def resetUwbPosNow(): Unit = {...}
}
```

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 test generated by ScalaCheck is a sequence of random commands acting on the system. The commands we have specified are *RunSegments* and *AddFault*, which have the following generators: *genRunSegments*, *genFaultAnchor*, *genFaultWheelSlip* and *genFaultYaw*. The first generator, *genRunSegments*, results in the generation of a random trajectory that is executed using the *RunSegments* command. The other generators are for injecting various kinds of faults into the system using the *AddFault* command, this is explained further in Section 3.1.

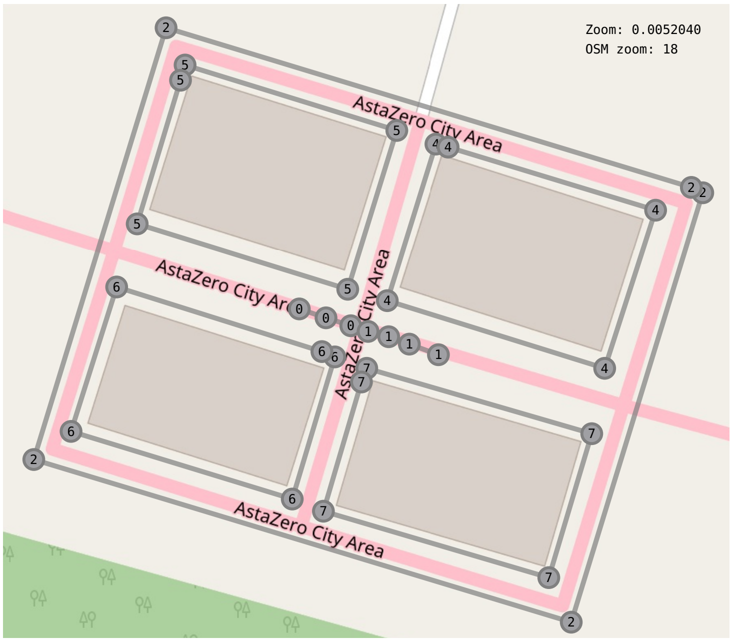


Figure 6: A driving area defined by route with ID 2 with the cutouts 4, 5, 6 and 7. Route 1 is the recovery route and route 0 is the start trajectory. The route IDs are written in the vertex circles of the polygons.

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 occured.

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: 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.

With these concepts established, the state maintained by ScalaCheck to facilitate test-case generation has the following content:

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

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) 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 postcondition 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 finishes 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 [5], 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 re-run 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[1][2][3].

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 re-run 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, 11]. 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:

```
void pos_uwb_update_dr(float imu_yaw, float turn_rad, float speed)
 {
  fi_inject_fault_float("uwb_travel_dist", &travel_dist);
```

[1]https://github.com/vedderb/rise_sdvp

[2]https://github.com/vedderb/rise_sdvp/tree/master/Linux/scala_test

[3]https://github.com/vedderb/rise_sdvp/tree/master/Linux/RControlStation

```
  fi_inject_fault_float("uwb_yaw", &imu_yaw);
  ...
}
```

These probes are controlled by our embedded C FaultCheck library, which is controlled from ScalaCheck using command generators such as:

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

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 distance 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 FaultCheck 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:

- 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.

- 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 [9] 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 to 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 illustrate 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 $\pm\theta°$ to the sides (line A and C), where $\theta = 30°$ and $L = 2\,\mathrm{m}$ 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, 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 until we 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

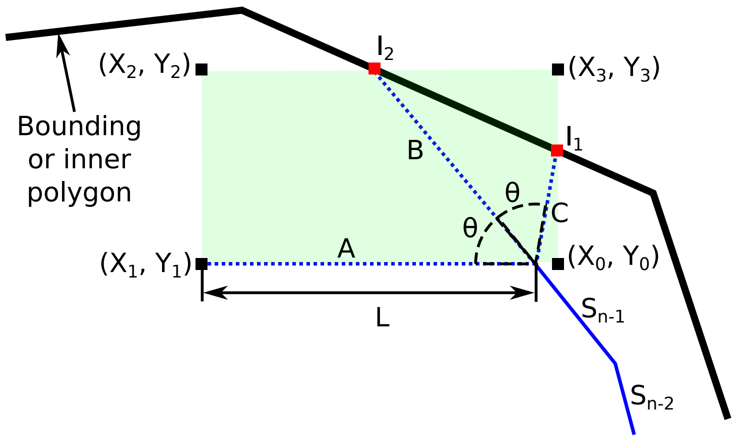


Figure 7: Rectangle considered for trajectory segment generation.

as it becomes increasingly 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. 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

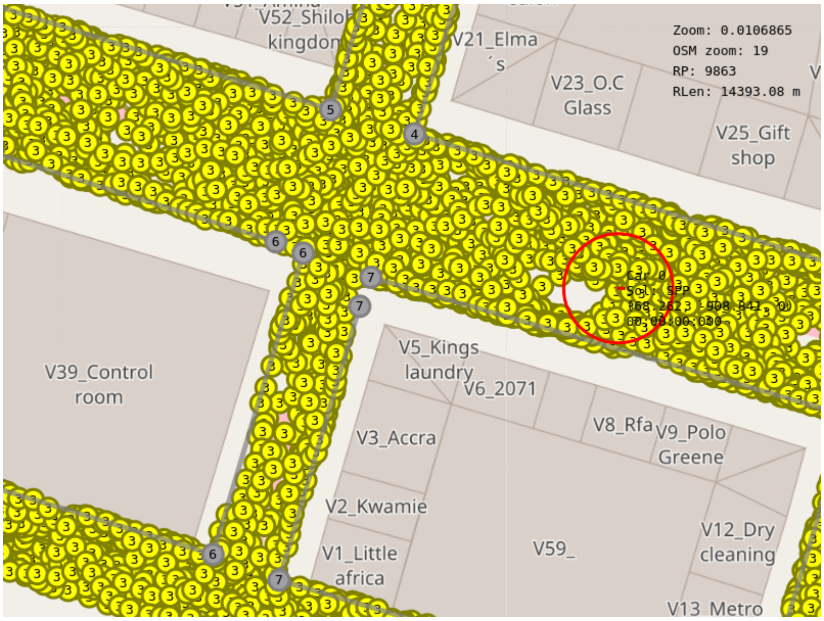


Figure 8: Part of a 14 km long trajectory generated within the driving area from Figure 6.

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:

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

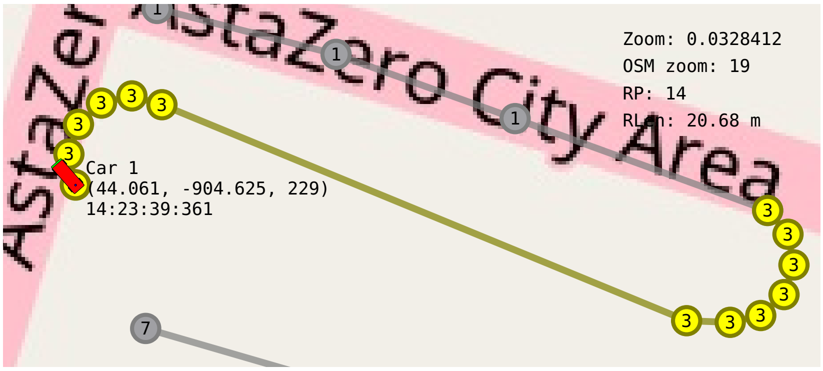


Figure 9: Trajectory generated by extending arcs from a starting point-orientation (the car) to a goal point-orientation (the recovery trajectory (trajectory 1)).

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.

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

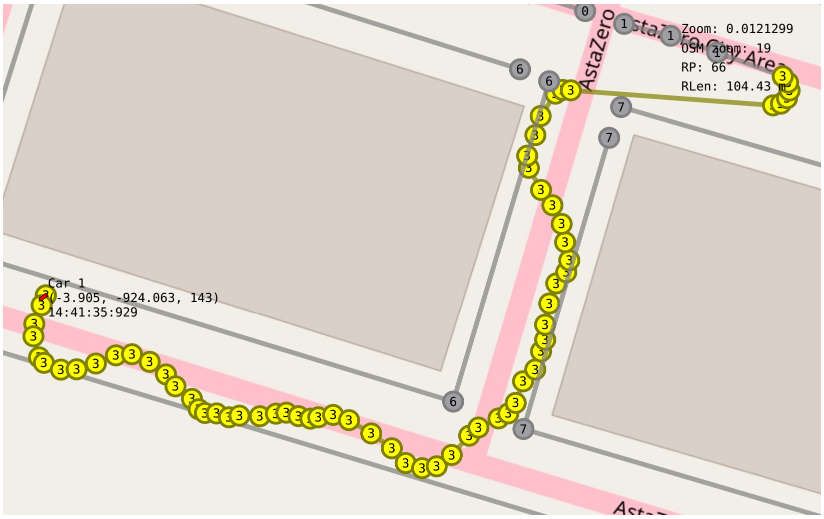


Figure 10: Recovery trajectory generated without optimization. Length: 104.4 m.

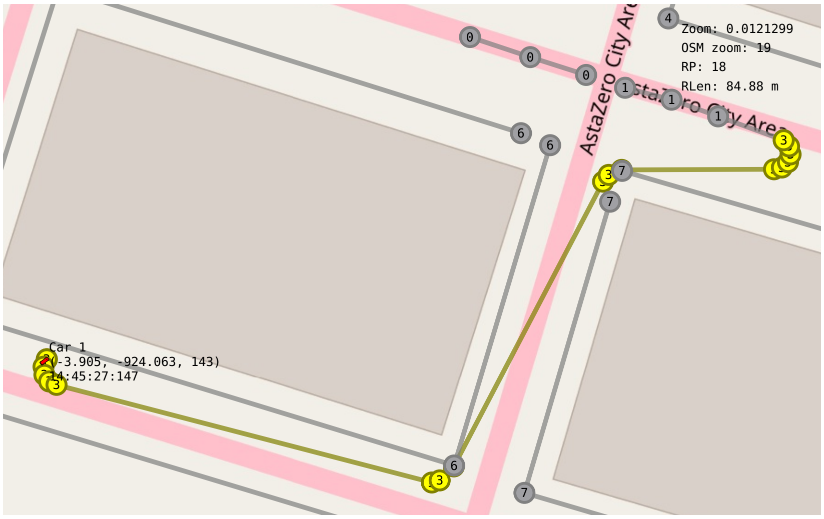


Figure 11: Trajectory from Figure 10 after the optimization pass. New length: 84.9 m.

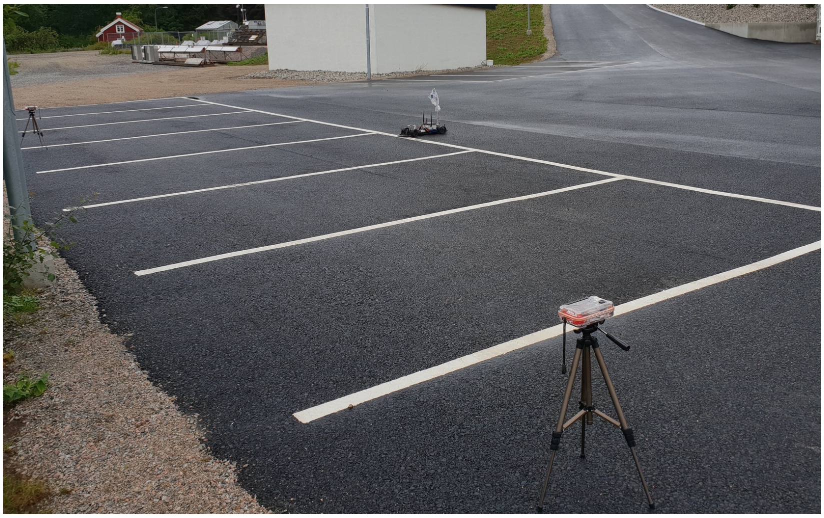


Figure 12: Our model car driving on a parking lot with two UWB anchors mounted on tripods in waterproof boxes.

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.

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

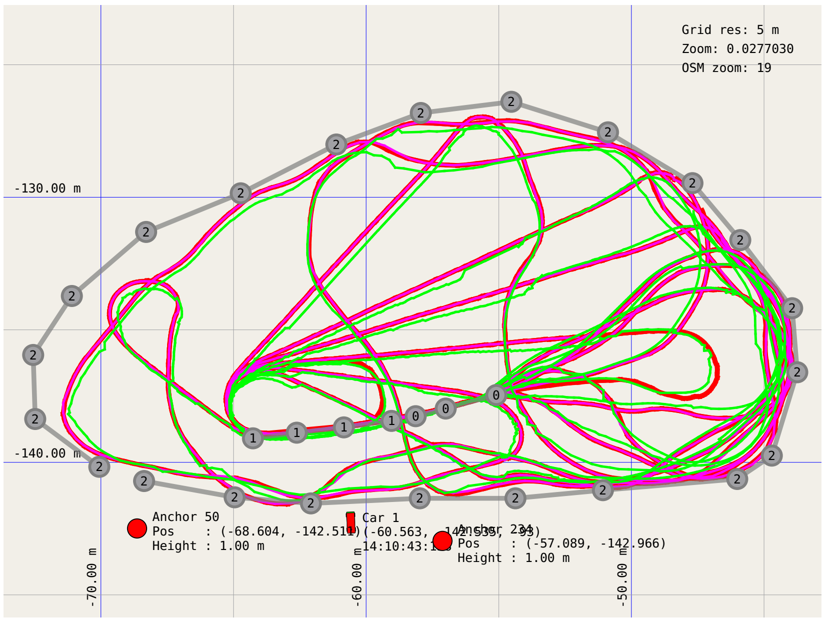


Figure 13: Traces after 10 test campaigns without FI. The RTK-SN traces are shown in magenta and the UWB traces are shown in green.

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.

Then we ran an experiment with FI enabled in the same area, with the results shown in Figure 14. The figure shows overlapping traces for re-running the same experiment three times using the return functionality described in Section 3.3. Figure 15 shows the maximum difference between the UWB and the RTK-SN positions for each RunSegment command for the three re-runs of the same experiment, as well as the difference between the individual experiments. The experiment consisted of the following commands:

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

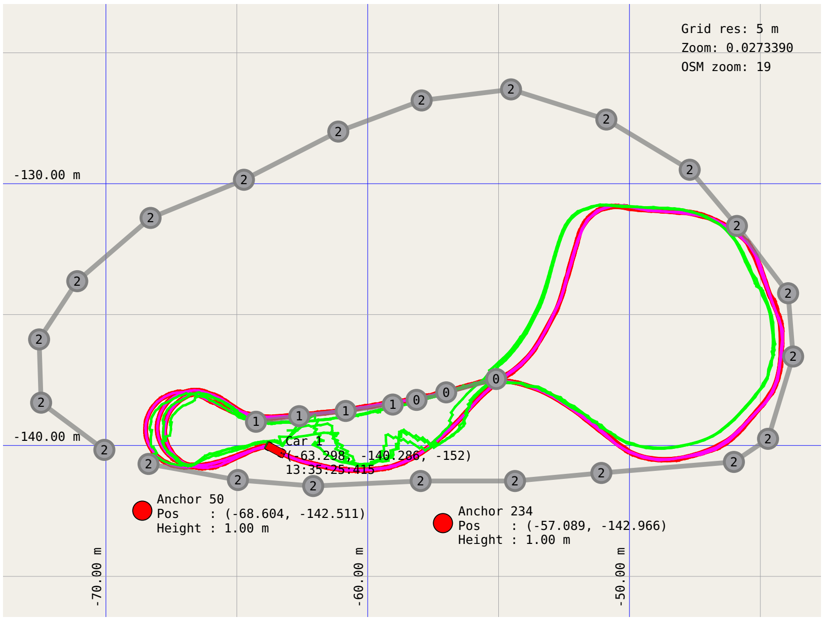


Figure 14: One test repeated three times with FI enabled. The traces for them are overlayed, and mostly overlapping. Maximum UWB deviation: 1.7 m

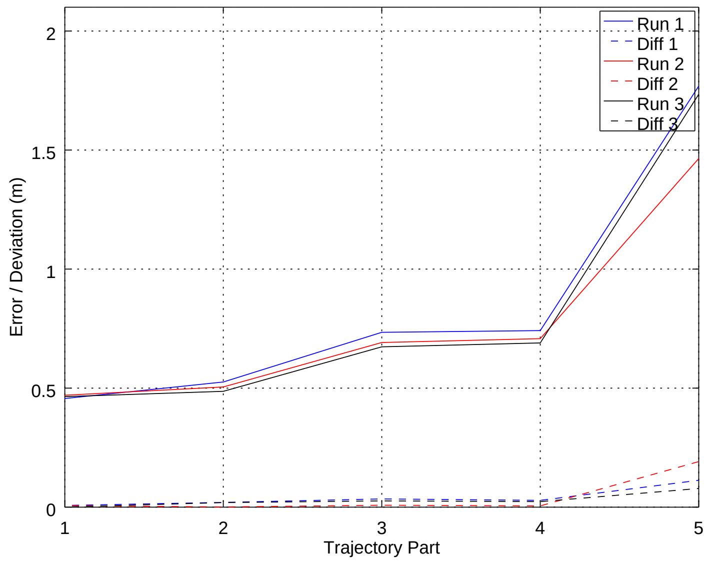


Figure 15: The difference between the UWB and RTK-SN positions for the three re-runs of the experiment shown in Figure 14. The dotted lines show the difference from each individual test to the average, indicating the repeatability.

```
   Reflection
10 AddFault(uwb_range_234,AMPLIFICATION,2.0,21,6) // UWB
   Reflection
11 RunSegment(...)
```

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. The combination of the later faults however brought the deviation up to around 1.5 m, which made the postcondition of the RunSegment command and thus the experiment fail. 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 re-ran the same experiment without FI enabled three times, as shown in Figure 16 and Figure 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 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 Figure 18 and Figure 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 re-runs of the experiment had consistent results, with less than 0.15 m difference between re-runs, 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,

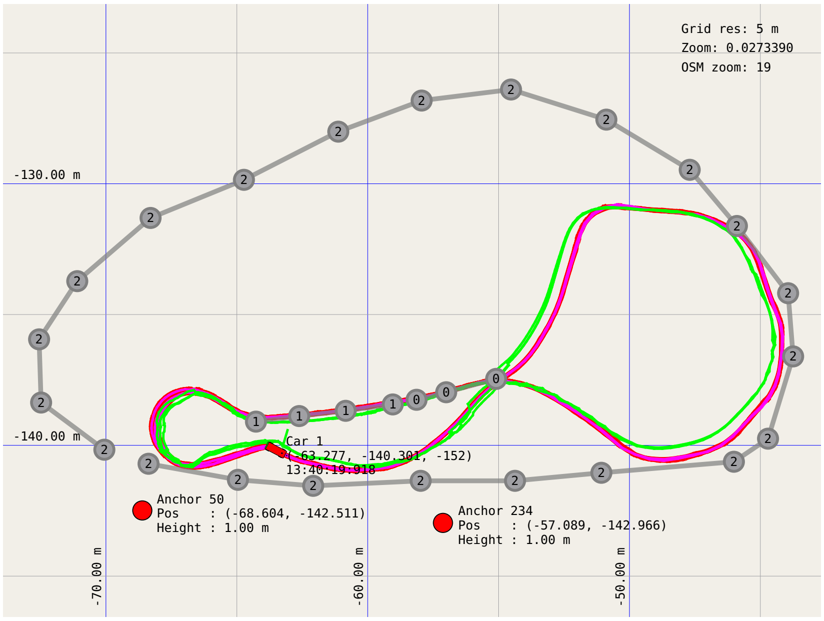


Figure 16: The same test as in Figure 14, but with FI disabled. The traces for repetition runs are overlayed, and mostly overlapping. Maximum UWB deviation: 0.55 m.

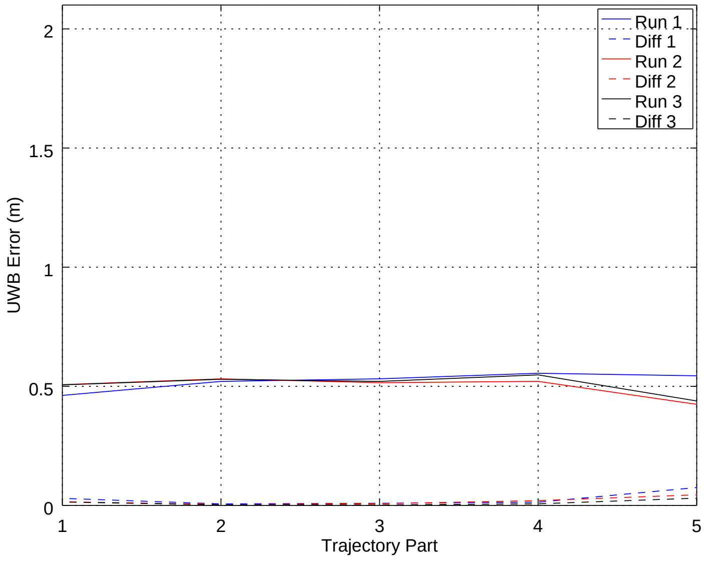


Figure 17: The difference between the UWB and RTK-SN positions for three re-runs of the experiment shown in Figure 16. The dotted lines show the difference from each individual test to the average, indicating the repeatability.

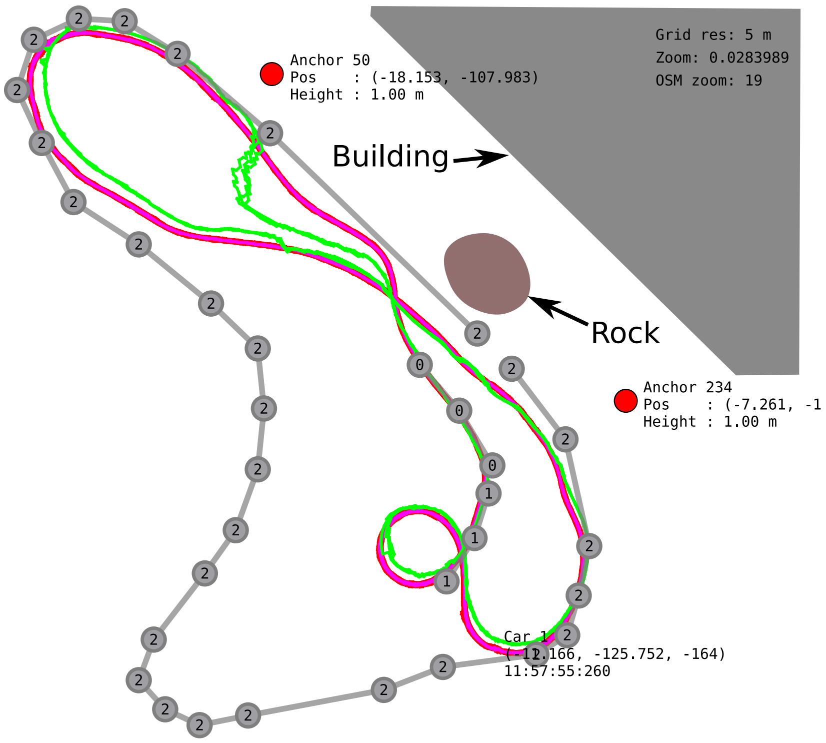


Figure 18: Experiment on parking lot with rock without FI, repeated three times. The traces for the repetitions are overlayed, and mostly overlapping. Maximum UWB deviation: 2 m.

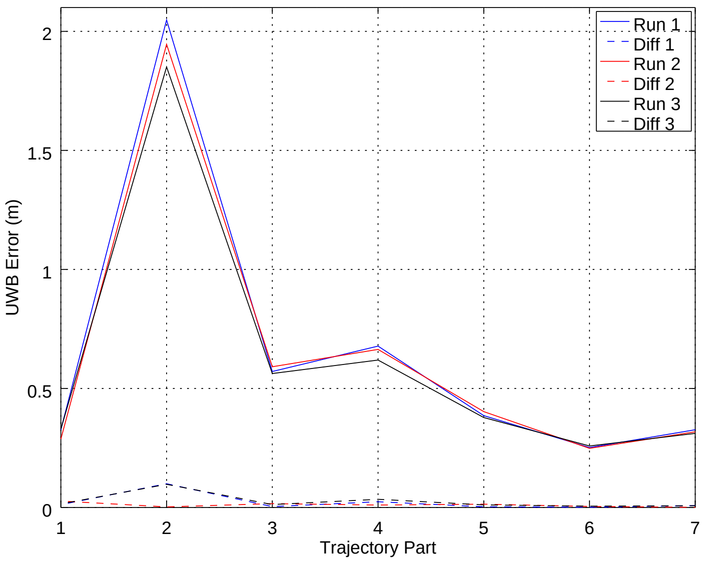


Figure 19: The difference between the UWB and RTK-SN positions for the three re-runs of the experiment shown in Figure 18. The dotted lines show the difference from each individual test to the average.

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 non-functional 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.

Acknowledgement

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: *IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2015, pp. 104–111.

[2] A. Soltani and F. Assadian. "A Hardware-in-the-Loop Facility for Integrated Vehicle Dynamics Control System Design and Validation". In: *7th IFAC Symposium on Mechatronic Systems MECHATRONICS 2016* 49.21 (2016), pp. 32–38.

[3] A. Mouzakitis, D. Copp, R. Parker, and K. Burnham. "Hardware-in-the-Loop System for Testing Automotive Ecu Diagnostic Software". In: *SAGE Measurement and Control Journal* 42.8 (2009), pp. 238–245.

[4] R. Nilsson. *ScalaCheck: The Definitive Guide*. Artima Press, 2014.

[5] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L. Fredlund, V. Gulias, J. Hughes, and S. Thompson. "Property-Based Testing - The ProTest Project". In: *Formal Methods for Components and Objects*. Ed. by F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 250–271.

[6] R. K. Iyer. "Experimental Evaluation". In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing*. FTCS'95. Pasadena, California: IEEE Computer Society, 1995, pp. 115–132.

[7] 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*. 1994, pp. 66–75.

[8] 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*. 2007, pp. 1–9.

[9] 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.* 1998, pp. 284–293.

[10] 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.* 2001, pp. 83–88.

[11] 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.* ES4CPS ’14. Dresden, Germany: ACM, 2014, 1:1–1:8.

[12] J. Cordes, J. Mössinger, W. Grote, and A. Lapp. ”Autosar standardised application interfaces”. In: *ATZautotechnology* 9.2 (Mar. 2009), pp. 42–45.

[13] M. Skoglund, T. Petig, B. Vedder, H. Eriksson, and E. M. Schiller. ”Static and dynamic performance evaluation of low-cost RTK GPS receivers”. In: *IEEE Intelligent Vehicles Symposium (IV).* June 2016, pp. 16–19.

[14] B. Vedder, J. Vinter, and M. Jonsson. ”A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving”. In: *Accepted for Publication in Hindawi Journal of Robotics.* 2018.

[15] M. Haklay and P. Weber. ”OpenStreetMap: User-Generated Street Maps”. In: *IEEE Pervasive Computing* 7.4 (Oct. 2008), pp. 12–18.

[16] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro. ”Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas”. In: *IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA).* Oct. 2016, pp. 7–12.

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

[18] B. Vedder, J. Vinter, and M. Jonsson. ”Accurate positioning of bicycles for improved safety”. In: *IEEE International Conference on Consumer Electronics (ICCE).* Jan. 2018, pp. 1–6.

[19] P. Olsson. ”Testing and Verification of Adaptive Cruise Control and Collision Warning with Brake Support by Using HIL Simulations”. In: *SAE Technical Paper.* SAE International, Apr. 2008.

[20] E. Bagalini and M. Violante. ”Development of an automated test system for ECU software validation: An industrial experience”. In: *15th Biennial Baltic Electronics Conference (BEC).* Oct. 2016, pp. 103–106.

[21] J. Yoon and C. D. Crane. ”Path planning for Unmanned Ground Vehicle in urban parking area”. In: *11th International Conference on Control, Automation and Systems.* Oct. 2011, pp. 887–892.

[22] M. S. M. Hashim and T.-F. Lu. ”Time-critical trajectory planning for a car-like robot in unknown environments”. In: *IEEE Business Engineering and Industrial Applications Colloquium (BEIAC).* Apr. 2013, pp. 836–841.