HALMSTAD
UNIVERSITY

Testing Safety-Critical Systems using Fault Injection and Property-Based Testing

Benjamin Vedder

LICENTIATE THESIS | Halmstad University Dissertations no. 13

Testing Safety-Critical Systems using Fault Injection and Property-Based Testing
© Benjamin Vedder
Halmstad University Dissertations no. 13
ISBN: 978-91-87045-29-5 (printed)
ISBN: 978-91-87045-28-8 (pdf)
Publisher: Halmstad University Press, 2015 | http://hh.se/hup
Printer: Media-Tryck, Lund

Abstract

Testing software-intensive systems can be challenging, especially when safety requirements are involved. Property-Based Testing (PBT) is a software testing technique where properties about software are specified and thousands of test cases with a wide range of inputs are automatically generated based on these properties. PBT does not formally prove that the software fulfils its specification, but it is an efficient way to identify deviations from the specification. Safety-critical systems that must be able to deal with faults, without causing damage or injuries, 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. The aim of this thesis is to investigate how knowledge and techniques from the areas of FI and PBT can be used together to test functional and safety requirements simultaneously.

We have developed a FI tool named FaultCheck that enables PBT tools to use common FI-techniques directly on source code. In order to evaluate and demonstrate our approach, we have applied our tool FaultCheck together with the commercially available PBT tool QuickCheck on a simple and on a complex system. The simple system is the AUTOSAR End-to-End (E2E) library and the complex system is a quadcopter simulator that we developed ourselves. The quadcopter simulator is based on a hardware quadcopter platform that we also developed, and the fault models that we inject into the simulator using FaultCheck are derived from the hardware quadcopter platform. We were able to efficiently apply FaultCheck together with QuickCheck on both the E2E library and the quadcopter simulator, which gives us confidence that FI together with PBT can be used to test and evaluate a wide range of simple and complex safety-critical software.

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. Finally, I would like to thank my colleagues at SP Technical Research Institute of Sweden and Halmstad University, the members of the PROWESS and KARYON EU Projects and the co-authors of the included papers for their support.

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).

List of Appended Papers

The following papers, referred to in the text by their Roman numerals, are included in this thesis.

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 practical tool named FaultCheck that enables property-based testing tools to use common fault injection techniques directly on C and C++ source code. The motivation for starting this work was that SP Technical Research Institute of Sweden, where Benjamin is employed, is supposed to contribute to the PROWESS EU project with knowledge about fault injection. PROWESS is about PROperty-based testing of WEb ServiceS. Benjamin has proposed and developed the tool FaultCheck 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. Accepted for 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 developed most of the electronics, written most of the software, developed a simulator for the platform 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. Accepted for 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 written the whole paper with some comments from his supervisors, derived fault models from the hardware platform and applied the testing platform presented in Paper I on the simulator.

List of Other Publications

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

PAPER IV: **Composable Safety-Critical Systems Based on Pre-Certified Software Components**
Andreas Söderberg and Benjamin Vedder, Proc. of ISSREW (2012).

PAPER V: **A Fault-Injection Prototype for Safety Assessment of V2X Communication**
Daniel Skarin, Benjamin Vedder, Rolf Johansson and Henrik Eriksson, Proc. of DEPEND (2014).

PAPER VI: **SafetyADD: A Tool for Safety-Contract Based Design**
Fredrik Warg, Benjamin Vedder, Martin Skoglund and Andreas Söderberg, Proc. of WoSoCer (2014).

Contents

Glossary

ADC	Analog to Digital Converter. 38
AHRS	Attitude and Heading Reference System. 39, 57
API	Application Programming Interface. 20
ASIL	Automotive Safety Integrity Level. 25
E2E	End-to-End. i, 6–11, 18, 21, 23, 25, 26, 29, 30
FFT	Fast Fourier Transform. 39
FI	Fault Injection. i, 3–7, 12, 17, 18, 20, 23, 30, 55, 56, 65, 67
GNSS	Global Navigation Satellite System. 35, 36, 50
GUI	Graphical User Interface. 45, 63, 65
IMU	Inertial Measurement Unit. 36–38, 50
ITS	Intelligent Transportation System. 45, 46, 62
LDM	Local Dynamic Map. 43, 60, 62
MAV	Micro Air Vehicle. 35, 36
PBT	Property-Based Testing. i, 1, 2, 4–8, 12, 17–21, 23, 24, 30, 55
PID	Proportional-Integral-Derivative. 40, 42, 58
PPM	Pulse-Position Modulation. 38
SLAM	Simultaneous Localization and Mapping. 37
SUT	System Under Test. 5, 20, 55, 56, 61, 65, 66, 69
ToF	Time of Flight. 37, 39

1. Introduction

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

1.1 Property-Based Testing

Writing software tests is often done in the form of unit tests, where each test case with fixed input data has to be written manually. In order to create a wide variety of test cases, a technique named Property-Based Testing (PBT) can be used [1]. 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 [2]. 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 [3], Erlang [4], Python [5] and Java [6]. ScalaCheck, which is the QuickCheck algorithm implemented in the Scala programming language, is integrated in the Scala testing framework ScalaTest [7] and used by prominent Scala projects such as the Akka concurrency framework [8]. One commercially available implementation of QuickCheck is Erlang QuickCheck [4], which has been used for the testing of large scale systems [9, 10]. Since we have been working in close collaboration with QuviQ[1], the company behind Erlang QuickCheck, we have been using Erlang QuickCheck in our experiments described later in this thesis.

1.1.1 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* 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 counter example, 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.

1.1.2 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

[1]http://quviq.com/

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 fails when the command *Decrement* is run when the counter 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.

1.2 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. Fault Injection (FI) is a method where the occurrence of faults is accelerated in order to exercise and evaluate fault handling mechanisms [11]. In safety standards such as the IEC 61508 [12], FI is mandatory in order to reach high diagnostics coverage. The automotive standard ISO 26262 [13] highly recommends FI to reach diagnostics coverage over 90%.

Traditional targets for FI have been hardware such as microprocessors and memories. One example of physical FI is scan-chain implemented FI [14] 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 [15]. 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 is often required to build special architectures that are designed to detect and handle such faults. For example, the fault tolerant version of the Leon microprocessor [16] has several internal fault detection mechanisms.

Another type of physical FI is pin-level FI [17] 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 mechanisms have to be used.

Physical FI, as presented above, can be emulated by injecting faults into models of hardware [18, 19] or directly into software and source code [20, 21]. FI can also be used on models of software (e.g. Simulink[1] models) that can be used to generate source code [22]. While emulating physical faults can be less realistic than doing physical FI

[1] http://mathworks.com/products/simulink/

on actual hardware, it has the advantage that it can be done early in the development process before the final hardware is finished. Other advantages are that setting up and repeating the same experiments can be easier.

In the scope of this thesis we are dealing with software testing, so we will be emulating hardware faults in software and simulations to evaluate the fault tolerance of software-intensive systems.

1.2.1 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 and 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, can be identified which cause unacceptable deviation from the golden run for the input vector of this test.

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 1.1.1) 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 without having to construct these runs manually.

1.2.2 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 VHDL model of a microprocessor has higher reachability than scan-chain implemented FI because the whole model can be accessed without restrictions. *Repeatability* describes how well the same experiment can be repeated. Heavy-ion FI is an example where it is very 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

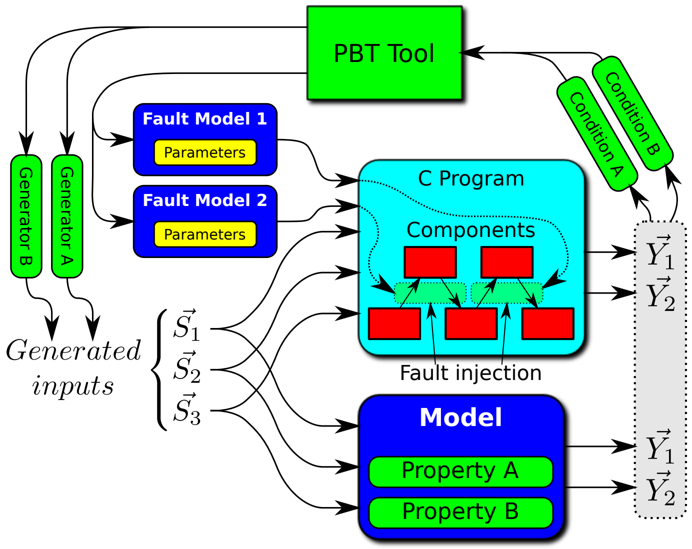


Figure 1: Property-based testing combined with fault injection inside a C program.

technique will have some intrusiveness on the source code that we are testing.

1.3 Aim

The aim of this thesis is to investigate how knowledge from the areas of PBT and FI can be combined to test functional and safety requirements simultaneously. A conceptual overview of how that could look like when done on a C-program can be seen in Figure 1. The idea is to do stateful PBT as described in Section 1.1.1, but with the addition of software-implemented FI on the System Under Test (SUT). Compared to doing FI alone, the PBT tool can automatically generate valid input stimuli and determine what the system state and output should be based on that input. This makes it possible to automatically generate thousands of golden runs and evaluate the functional properties of programs under the influence of emulated hardware faults. Our approach should bring advantages for both the PBT and the FI communities.

1.4 Research Questions

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

Q2 How can realistic fault models be derived for a system and how can they be represented and injected using a testing platform based on PBT and FI?

1.5 Research Approach

We have created a practical testing platform that uses techniques from the areas of PBT and FI. Our testing platform is based on the PBT tool Erlang QuickCheck and a FI tool that we developed ourselves, named FaultCheck. FaultCheck enables PBT tools such as QuickCheck to inject common fault models into C and C++ programs.

First we apply our testing platform on a small piece of software, namely the AUTOSAR End-to-End (E2E)-library. With this, we demonstrate that our concept works. After that, we develop a significantly more complex system that we test using our testing platform, namely a quadcopter simulator. By testing both the E2E-library and the simulator efficiently, we show that our testing platform, based on FaultCheck and QuickCheck, scales for testing of both simple and complex systems.

1.6 Research Contributions

C1 We present ideas 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).

C2 We develop a hardware quadcopter system with a novel approach for localization and collision avoidance. In order to test functional requirements and fault tolerance for the quadcopters, we develop a simulator with a strong connection to the hardware platform. We test the simulator using our testing platform based on FaultCheck together with QuickCheck (Paper II and Paper III).

C3 We show how to derive realistic fault models from our quadcopter platform and how to represent and inject them in our simulator using our testing platform based on FaultCheck and QuickCheck (Paper III). This gives us confidence that fault models can be derived in a similar way for a wide range of complex systems from different domains.

C4 We show how our testing platform scales by using it both on a simple E2E-library (Paper I), and on a significantly more complex system, namely a quadcopter simulator (Paper II and Paper III).

2. Summary of Papers

Included in this thesis are three papers that summarize 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 from PBT tools. In Paper II we present a novel hardware quadcopter platform and a simulator that we have developed with our testing platform in consideration. Finally, in Paper III, we present details about how we tested our quadcopter simulator using FaultCheck and QuickCheck.

2.1 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 2. 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 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 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 [23] as an example application and the PBT tool Erlang QuickCheck to control FaultCheck, as shown in Figure 3.

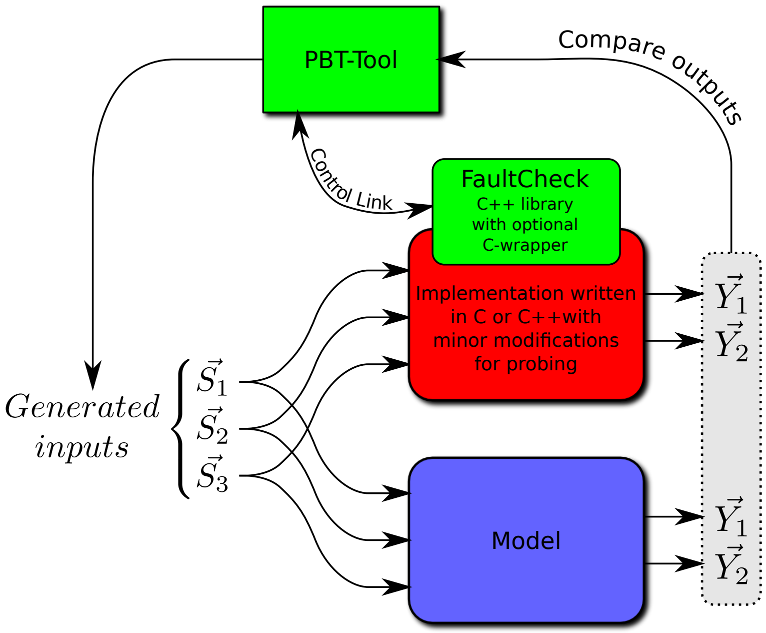


Figure 2: FaultCheck used together with a PBT-tool.

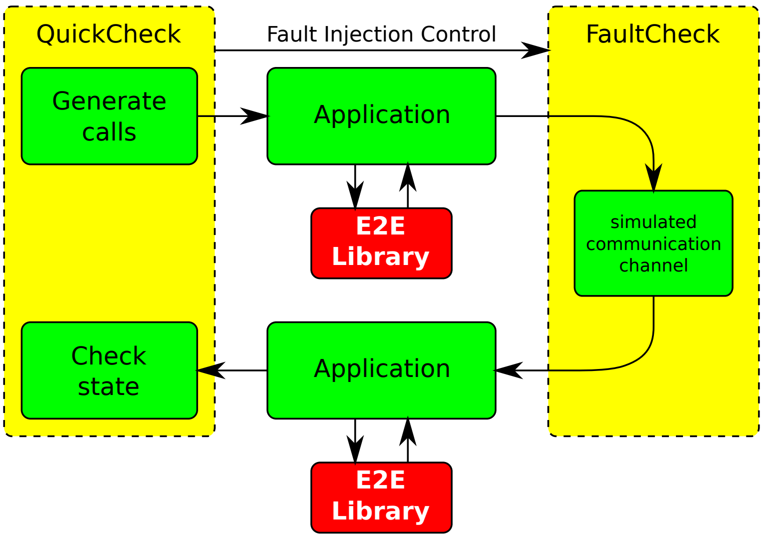
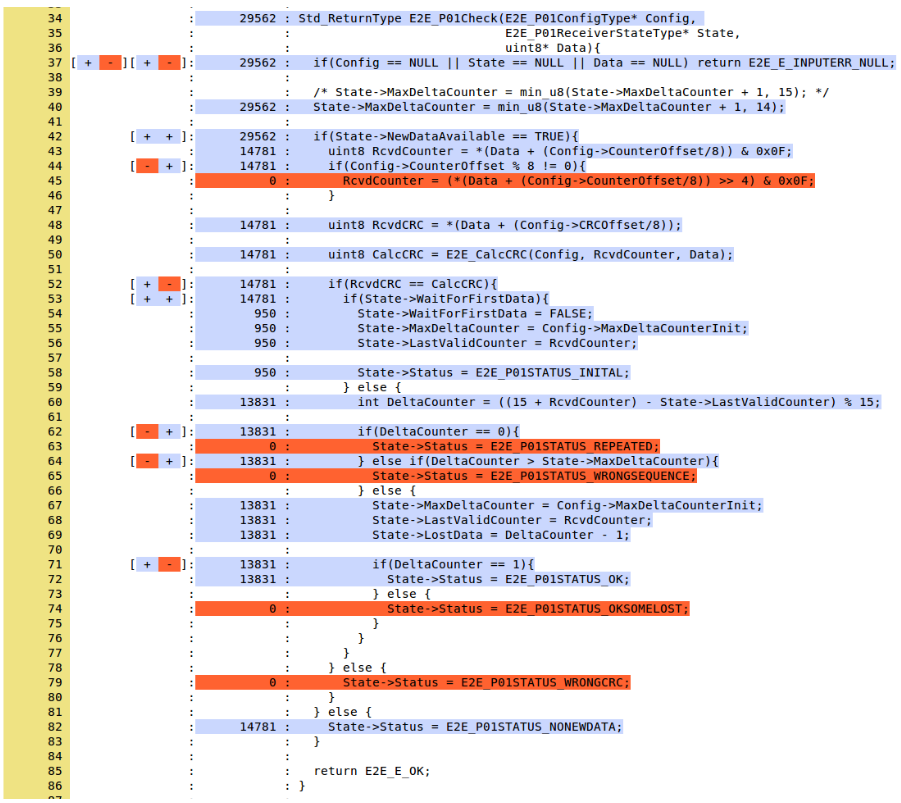


Figure 3: Experiment setup to evaluate the E2E library.



```
                 :   29562 : Std_ReturnType E2E_P01Check(E2E_P01ConfigType* Config,
                 :         :                             E2E_P01ReceiverStateType* State,
                 :         :                             uint8* Data){
[ + - ][ + - ]:   29562 :   if(Config == NULL || State == NULL || Data == NULL) return E2E_E_INPUTERR_NULL;
                 :         :
                 :         :   /* State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 15); */
                 :   29562 :   State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 14);
                 :         :
        [ + + ]:   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;
                 :         :     }
                 :         :   } else {
                 :   14781 :     State->Status = E2E_P01STATUS_NONEWDATA;
                 :         :   }
                 :         :
                 :         :   return E2E_E_OK;
                 :         : }
```

Figure 4: The code coverage of the E2E library when running the experiment without injecting any faults.

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 4 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 5 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 6 shows that when the experiment is run with all fault injection active, 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.

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

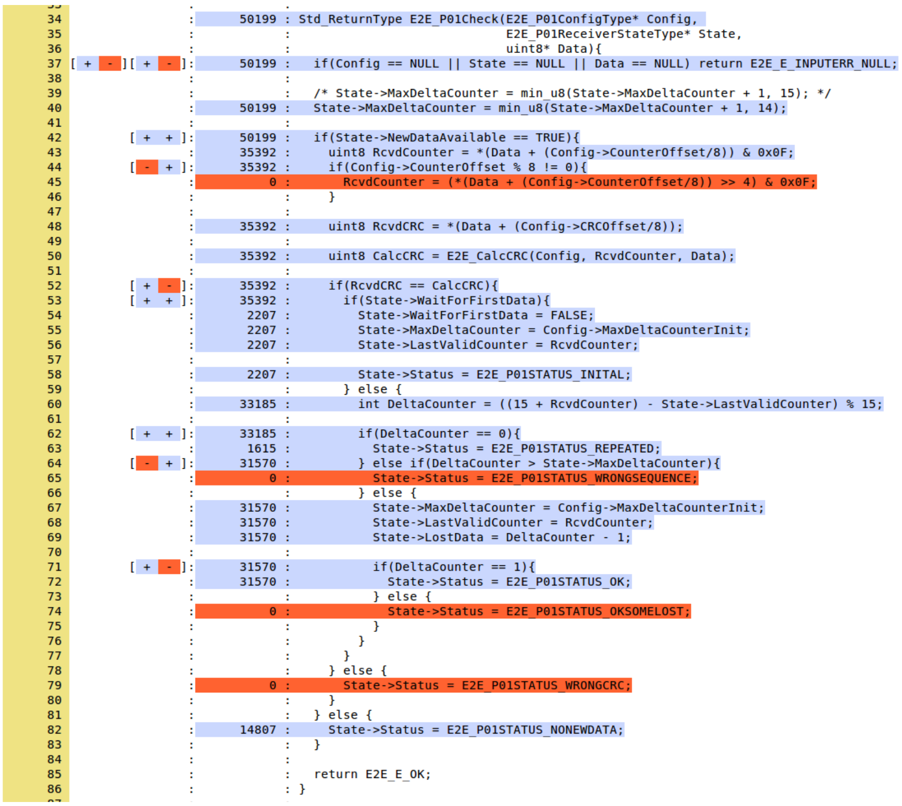
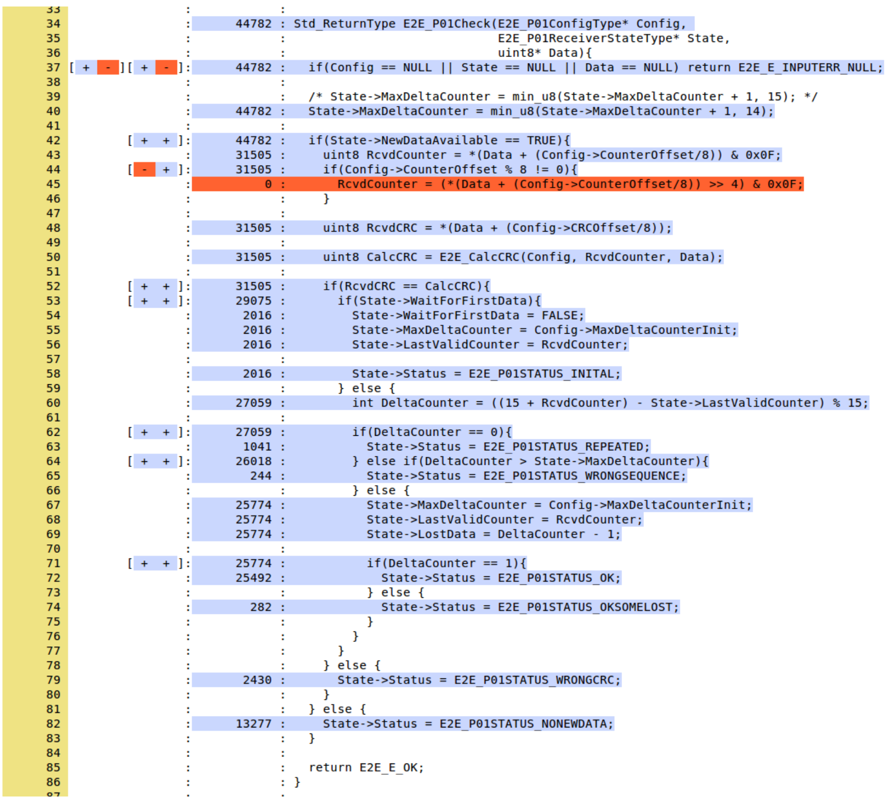

```
                 :       50199 : Std_ReturnType E2E_P01Check(E2E_P01ConfigType* Config,
                 :             :                             E2E_P01ReceiverStateType* State,
                 :             :                             uint8* Data){
[ + - ][ + - ]:       50199 :   if(Config == NULL || State == NULL || Data == NULL) return E2E_E_INPUTERR_NULL;
                 :             :
                 :             :   /* State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 15); */
                 :       50199 :   State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 14);
                 :             :
         [ + + ]:       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;
                 :             :     }
                 :             :   } else {
                 :       14807 :     State->Status = E2E_P01STATUS_NONEWDATA;
                 :             :   }
                 :             :
                 :             :   return E2E_E_OK;
                 :             : }
```

Figure 5: The code coverage of the E2E library when running the experiment when only injecting repetition communication faults.



```
                  :   44782 : Std_ReturnType E2E_P01Check(E2E_P01ConfigType* Config,
                  :         :                             E2E_P01ReceiverStateType* State,
                  :         :                             uint8* Data){
[ + - ][ + - ]:   44782 :   if(Config == NULL || State == NULL || Data == NULL) return E2E_E_INPUTERR_NULL;
                  :         :
                  :         :   /* State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 15); */
                  :   44782 :   State->MaxDeltaCounter = min_u8(State->MaxDeltaCounter + 1, 14);
                  :         :
        [ + + ]:   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;
                  :         :     }
                  :         :   } else {
                  :   13277 :     State->Status = E2E_P01STATUS_NONEWDATA;
                  :         :   }
                  :         :
                  :         :   return E2E_E_OK;
                  :         : }
```

Figure 6: The code coverage of the E2E library when running the experiment while injecting all possible communication faults.

2.2 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. [24, 25] 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 also 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.

2.3 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.

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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).

[11] 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.

[12] International Electrotechnical Commission. *IEC 61508: Functional safety of electrical/-electronic/programmable electronic safety related systems*. Norm. 2010.

[13] International Organization for Standardization ISO. *ISO 26262: Road vehicles – Functional safety*. Norm. 2011.

[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. 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] S. Han, K.G. Shin, and H.A. 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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 2 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 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.

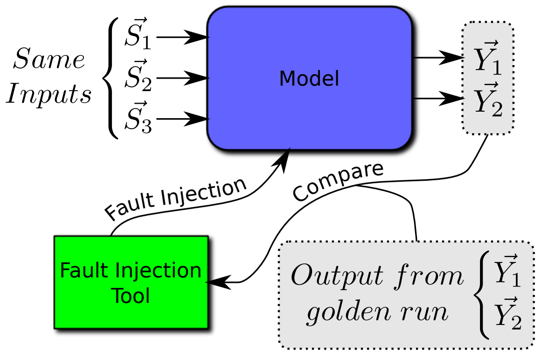


Figure 1: One example of model-based fault injection.

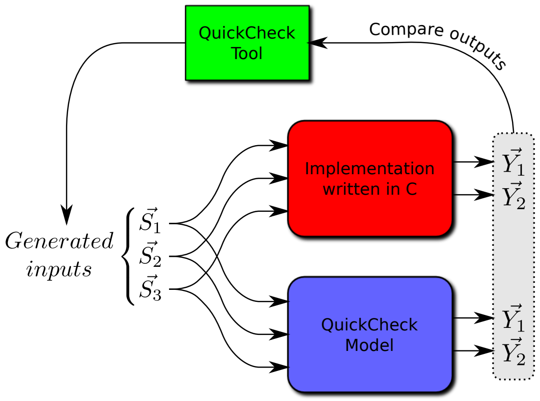


Figure 2: One example of PBT.

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.

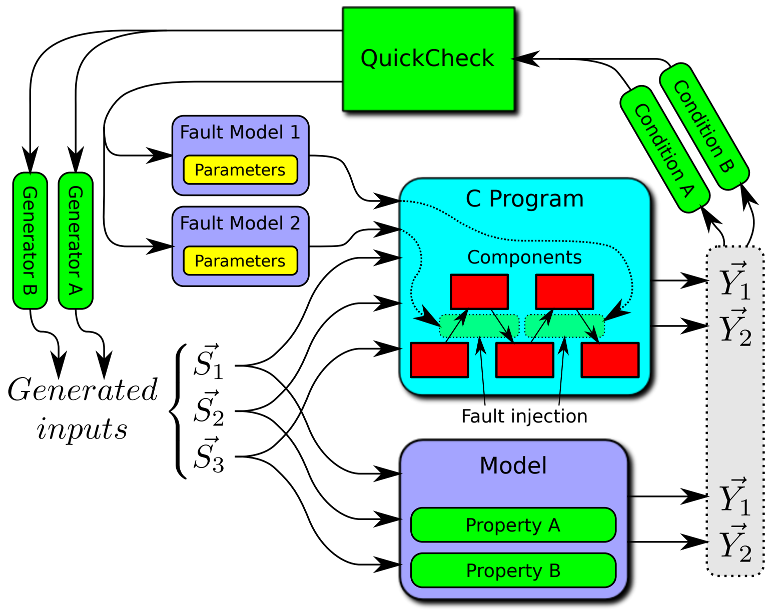


Figure 3: Property-based testing combined with fault injection inside a C program.

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.

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,

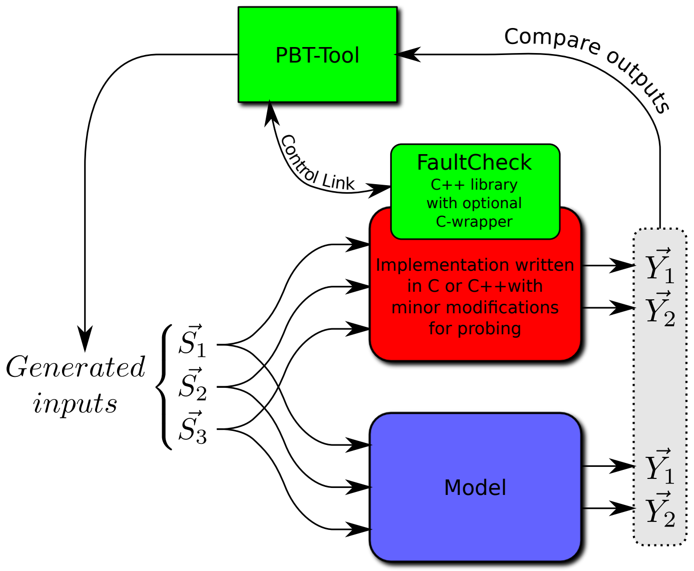


Figure 4: The FaultCheck probing-library controlled from a PBT-tool.

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

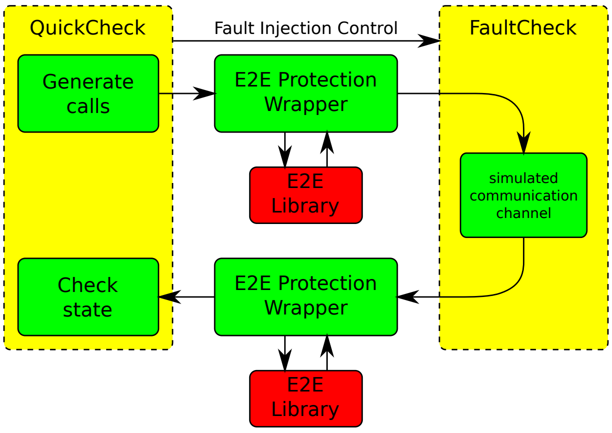


Figure 5: Experiment set-up of the E2E evaluation system.

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.

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.C. Fabre, J.C. 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.G. Silva. "RIFLE: A General Purpose Pin-Level Fault Injector". In: *Dependable Computing — EDCC-1*. Ed. by Klaus Echtle, Dieter Hammer, and David 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, Ravishankar K. Iyer, and Luke 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.G. Shin, and H.A. 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, João Gabriel S., and Dep Engenharia 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.Å. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects*. Ed. by FrankS. Boer, MarcelloM. Bonsangue, Stefan Hallerstede, and Michael 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] International Organization for Standardization ISO. *ISO 26262: Road vehicles – Functional safety*. Norm. 2011.

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

[27] PROWESS 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

Accepted for Publication 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 8 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 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

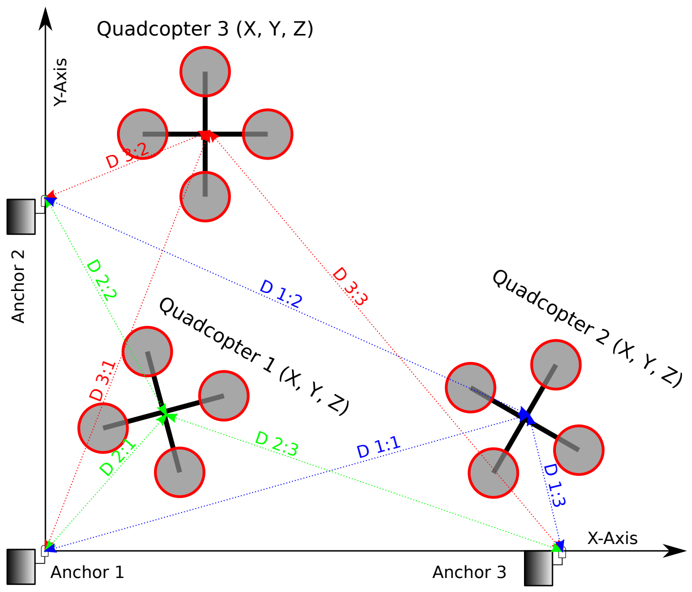


Figure 1: The ultrasound-localization system with anchors. The measured distances from the copters to the anchors are marked D a:b.

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.

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.

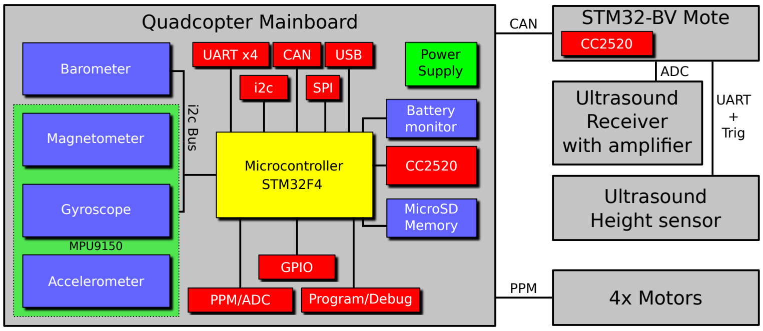


Figure 2: The hardware setup on each quadcopter.

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.

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

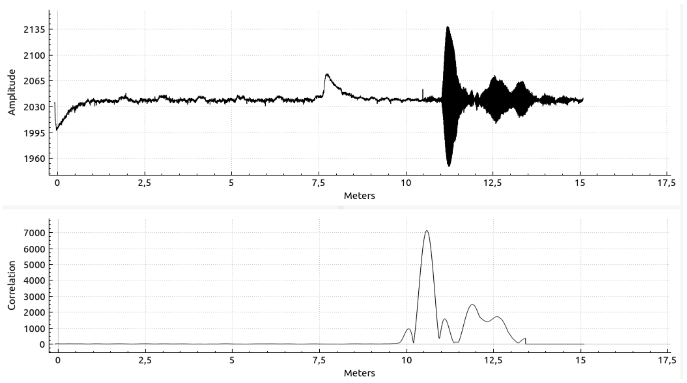


Figure 3: Ultrasound samples recorded for a time corresponding to 10 meters and the cross correlation result.

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.

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

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

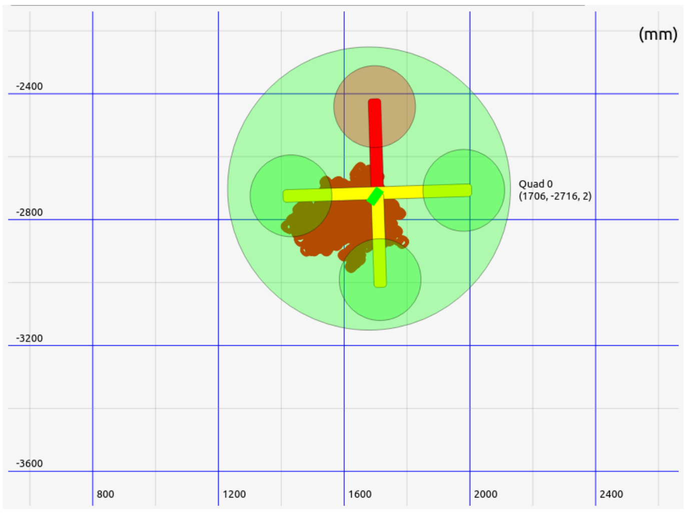


Figure 4: The estimated position during a 60 s long hover.

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 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

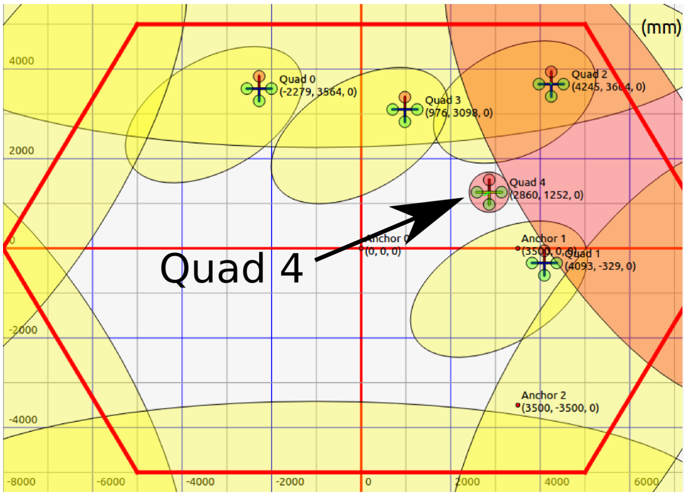


Figure 5: A screenshot of the risk contours from the perspective of Quad 4. The red contour is red because there is an overlap between the copters own comfort zone (Quad 4) and the risk contour around the right upper wall.

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, 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-

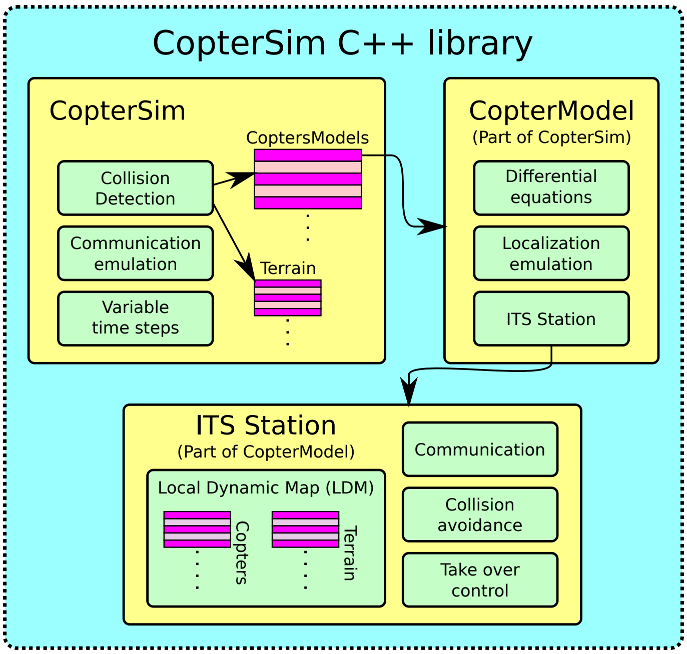


Figure 6: The simulation architecture. CopterSim has a list with all copters and checks for collisions between them and the terrain in every iteration. CopterModel handles the physical model of every copter and has an ITS-station that handles collision avoidance. The localization emulation is also part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

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 $\pm 15°$.
- The pitch output, chosen between $\pm 15°$.

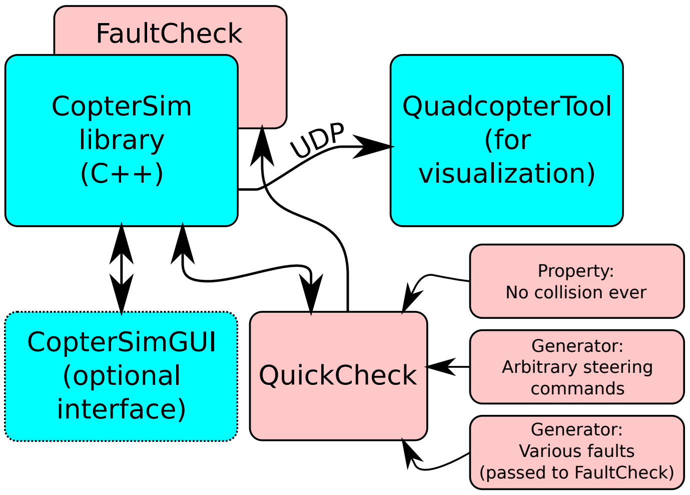

FaultCheck
CopterSim library (C++)
UDP
QuadcopterTool (for visualization)
Property: No collision ever
CopterSimGUI (optional interface)
QuickCheck
Generator: Arbitrary steering commands
Generator: Various faults (passed to FaultCheck)

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

- The yaw rate output, randomly chosen between $\pm 90^\circ$ 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 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

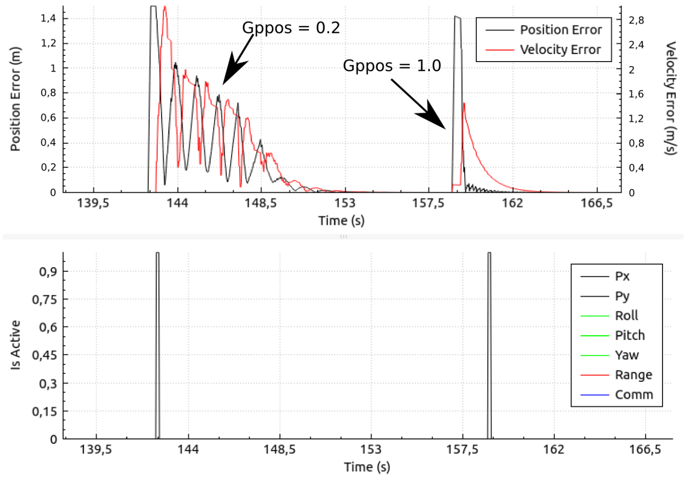


Figure 8: Fault injection with 1.5 m position offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,pos} = 0.2$ and the second one had $G_{p,pos} = 1.0$.

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 $G_{p,vel}$.

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.

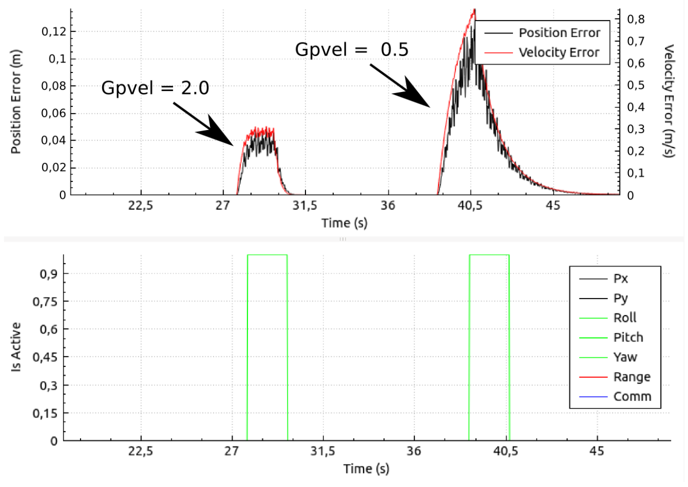


Figure 9: Fault injection with 5° pitch offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 2.0$ and the second one had $G_{p,vel} = 0.5$.

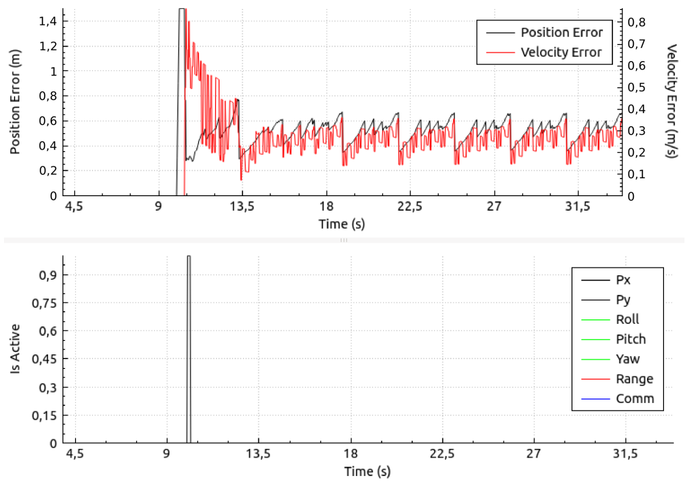


Figure 10: Fault injection with 1.5 m position offset and $G_{p,pos} = 2.0$. The lower part shows when the faults are active and the upper part shows the position and velocity errors.

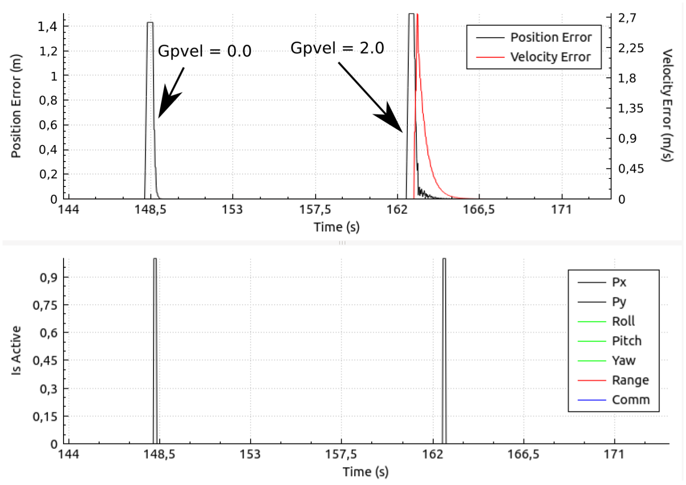


Figure 11: Fault injection with 1.5 m position offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 0.0$ and the second one had $G_{p,vel} = 2.0$.

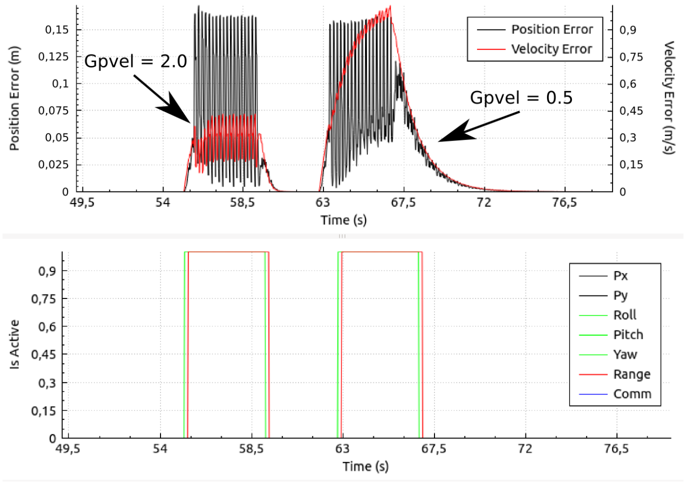


Figure 12: Fault injection with 5° pitch offset and 0.5 m anchor distance offset. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,vel} = 2.0$ and the second one had $G_{p,vel} = 0.3$.

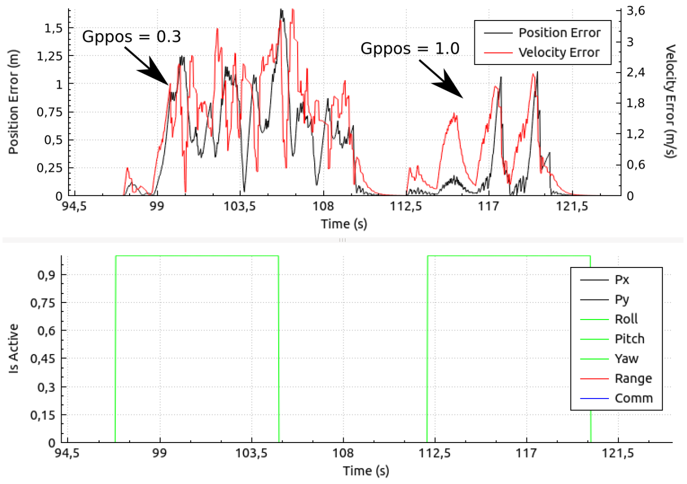


Figure 13: Fault injection with 0.95 pitch amplification. The lower part shows when the faults are active and the upper part shows the position and velocity errors. The first activation had $G_{p,pos} = 0.3$ and the second one had $G_{p,pos} = 1.0$.

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.

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.C. Fabre, J.C. 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.Å. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects*. Ed. by FrankS. Boer, MarcelloM. Bonsangue, Stefan Hallerstede, and Michael 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.G. Rajnarayan, S.L. Waslander, D. Dostal, Jung Soon Jang, and C.J. Tomlin. ”The Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control (STARMAC)”. In: *The 23rd Digital Avionics Systems Conference, DASC 04*. Vol. 2. 2004, pages.

[13] J. Pestana, J.L. 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] James 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.O.H. 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. Östberg 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

Accepted for Publication 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 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.

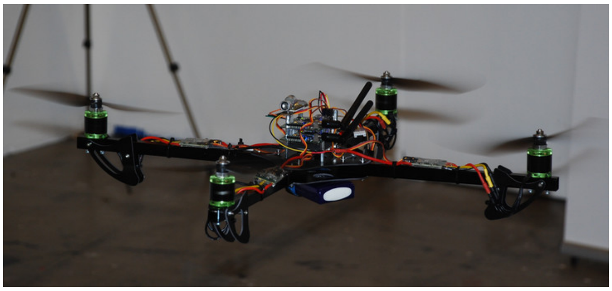


Figure 1: A photo of one of the quadcopters.

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

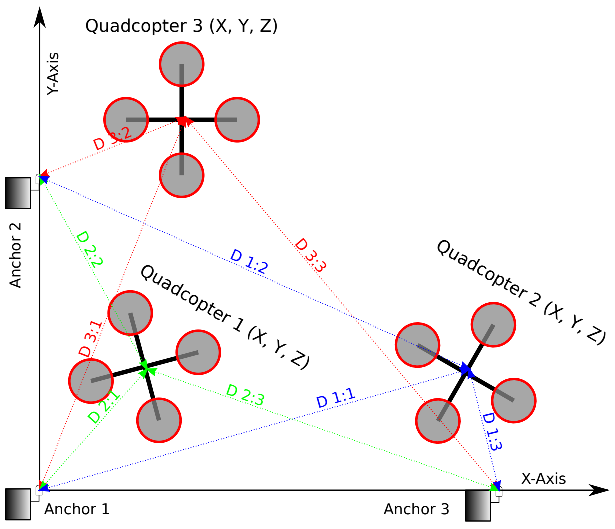


Figure 2: The ultrasound-localization system with anchors. The measured distances from the copters to the anchors are marked D a:b.

$[\theta_r(n), \theta_p(n), \theta_y(n)]^T$ 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:

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

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

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

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.

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.

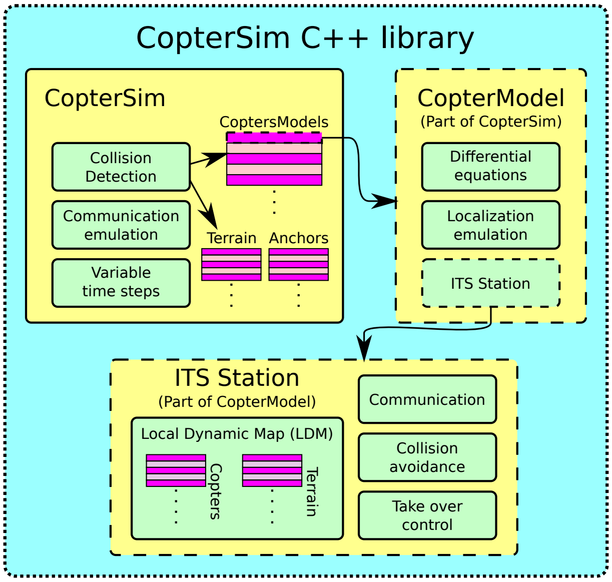
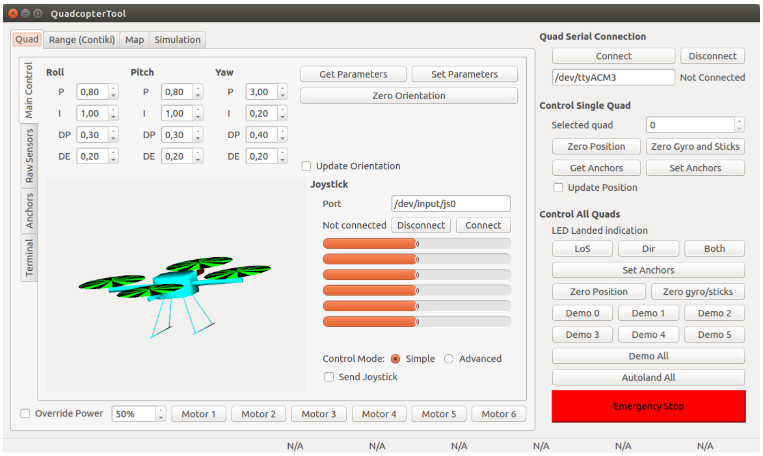


Figure 3: The simulator library. CopterSim has a list with all copters and checks for collisions between them and the terrain in every iteration. CopterModel handles the physical model of every copter and has an ITS-station that handles collision avoidance.

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



	Roll	Pitch	Yaw
P	0,80	0,80	3,00
I	1,00	1,00	0,20
DP	0,30	0,30	0,40
DE	0,20	0,20	0,20

Port /dev/input/js0
Not connected
Control Mode: [✓ Simple] Advanced
Override Power 50%
/dev/ttyACM3 Not Connected
Selected quad 0
N/A N/A N/A N/A N/A N/A

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.

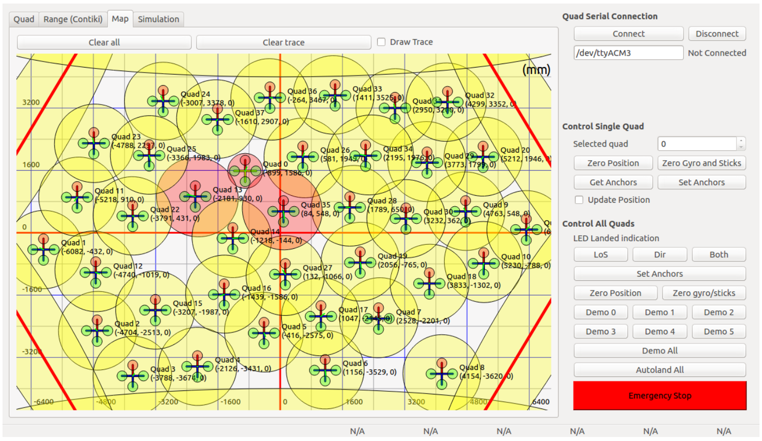
3.1 QuadcopterTool

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.

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



/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).

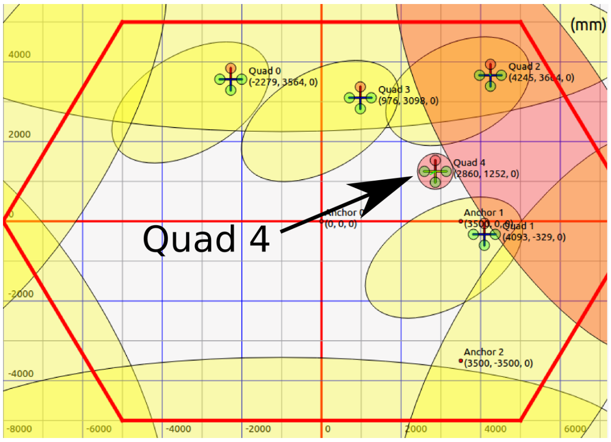
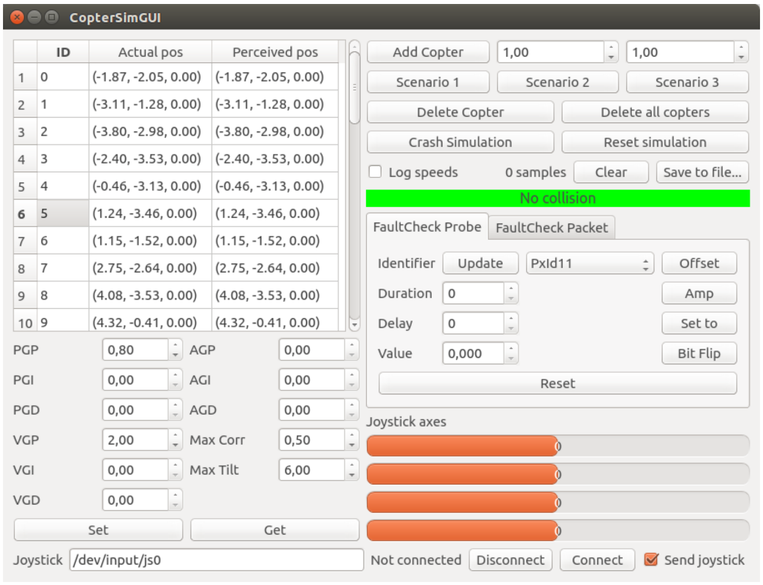


Figure 6: A screenshot of the risk contours from the perspective of Quad 4. The red contour is red because there is an overlap between the copter's own comfort zone (Quad 4) and the risk contour around the right upper wall.


CopterSimGUI

	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		

Joystick /dev/input/js0 Not connected [✓ Send joystick]

Add Copter 1,00 1,00

Log speeds 0 samples

No collision

Identifier PxId11

Duration 0

Delay 0

Value 0,000

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.

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 Quadcopter-Tool 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.

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 $\pm 15°$, 3) the pitch output, randomly chosen between $\pm 15°$, and 4) the yaw rate output, randomly chosen between $\pm 90°$ 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

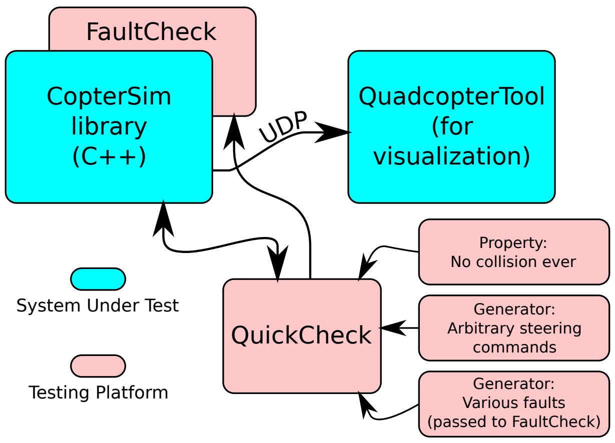

FaultCheck
CopterSim library (C++)
UDP
QuadcopterTool (for visualization)
Property: No collision ever
Generator: Arbitrary steering commands
Generator: Various faults (passed to FaultCheck)
QuickCheck
System Under Test
Testing Platform

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

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)]).

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, Ravishankar K. Iyer, and Luke 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.P.E. 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.C. Fabre, J.C. 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.G. Silva. ”RIFLE: A General Purpose Pin-Level Fault Injector”. In: *Dependable Computing — EDCC-1*. Ed. by Klaus Echtle, Dieter Hammer, and David 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.Å. Fredlund, V. Gulias, J. Hughes, and S. Thompson. ”Property-Based Testing - The ProTest Project”. In: *Formal Methods for Components and Objects*. Ed. by FrankS. Boer, MarcelloM. Bonsangue, Stefan Hallerstede, and Michael 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.O.H. 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] Jiaying 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.