This is the accepted version of a paper presented at 1st International Workshop on Safety and Security of Intelligent Vehicles (SSIV), Rio de Janeiro, Brazil, June 22, 2015.

Citation for the original published paper:


N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-28172
Using Simulation, Fault Injection and Property-Based Testing to Evaluate Collision Avoidance of a Quadcopter System

Benjamin Vedder  
Department of Electronics,  
SP Technical Research  
Institute of Sweden  
Email: benjamin.vedder@sp.se

Jonny Vinter  
Department of Electronics,  
SP Technical Research  
Institute of Sweden  
Email: jonny.vinter@sp.se

Magnus Jonsson  
School of Information Science,  
Computer and Electrical Engineering.  
Halmstad University  
Email: magnus.jonsson@hh.se

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.

I. 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 fulfills 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 platform1 described in Section II. 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 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

1https://www.youtube.com/watch?v=vk9Nrem9ZAE
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 copters. 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 copters. Since the visualization is created in real-time based on a list of QuickCheck commands (e.g. steering and F1) that lead to the collision, we can adjust and enhance the collision-avoidance mechanism and replay and visualize the experiment with the same commands over and over again attempting to avoid a collision caused by this series of commands.

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

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

II. QUADCOPTER SYSTEM

The hardware quadcopter platform that our simulator is based on consists of four copters, 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 copter 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:

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

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

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

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

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

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

\[
\begin{align*}
\frac{dx(n)}{dt} &= 9.82\tan(\theta_y(n) + \theta_{rofs}(n))dt + 9.82\tan(\theta_y(n) + \theta_{pofs}(n))dt \\
\frac{dy(n)}{dt} &= \left[ \begin{array}{c} c_y \\ s_y \end{array} \right] \\
\frac{dz(n)}{dt} &= \left[ \begin{array}{c} \cos(\theta_y(n)) \\ \sin(\theta_y(n)) \end{array} \right] \\
\end{align*}
\]

(3)

(4)

(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 \([\theta_x(n), \theta_y(n)]^T\) are 90° is not an issue because these angles are limited at 45°. This is then used to update the position \([P_x(n), P_y(n)]^T\):

\[
\begin{align*}
P_x(n) &= P_x(n - 1) + V_x(n)dt \\
P_y(n) &= P_y(n - 1) + V_y(n)dt \\
\end{align*}
\]

(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:

\[
\begin{align*}
\frac{dx(n)}{dt} &= \left[ P_x(n - 1) - P_x(\text{anchor}) \\ P_y(n - 1) - P_y(\text{anchor}) \\ P_z(n - 1) - P_z(\text{anchor}) \right] \\
d_a(n) &= \sqrt{(dx(n))^2 + (dy(n))^2 + (dz(n))^2} \\
err(n) &= d_a(n) - d_{\text{measured}}(n) \\
F_c(n) &= \frac{err(n)}{d_a(n)} \\
\end{align*}
\]

where \([P_x(n), P_y(n), P_z(n)]^T\) is the position of the copter, \([P_x(\text{anchor}), P_y(\text{anchor}), P_z(\text{anchor})]^T\) is the position of the anchor this measurement came from and \([d_{xz}(n), d_{zy}(n), d_{az}(n)]^T\) is the difference between them. Further, \(d_a(n)\) is the magnitude of the calculated difference, \(d_{\text{measured}}(n)\) 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 \([d_{xz}(n), d_{zy}(n), d_{az}(n)]^T\) 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 III has been an important aid for doing that.

\[
\begin{align*}
P_{xpos}(n) &= \frac{P_x(n)F_cG_{p,pos}}{d_{xyn}(n)F_cG_{p,pos}} \\
P_{ypos}(n) &= \frac{P_y(n)F_cG_{p,pos}}{d_{yn}(n)F_cG_{p,pos}} \\
D_{xpos}(n) &= \frac{(d_{xyn}(n)F_c - d_{xyn}(n - 1)F_c)G_{d,pos}}{d_{yn}(n)F_c - d_{yn}(n - 1)F_c} \\
D_{ypos}(n) &= \frac{(d_{yn}(n)F_c - d_{yn}(n - 1)F_c)G_{d,pos}}{d_{yn}(n)F_c - d_{yn}(n - 1)F_c} \\
\end{align*}
\]

(11)

(12)

Then, apply this to the position:

\[
\begin{align*}
P_x(n) &= \left[ P_x(n - 1) + P_{xpos}(n) + D_{xpos}(n) \right] \\
P_y(n) &= \left[ P_y(n - 1) + P_{ypos}(n) + D_{ypos}(n) \right] \\
\end{align*}
\]

(13)

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

\[
\begin{align*}
V_x(n) &= \left[ V_x(n - 1) + d_{xyn}(n)F_cG_{p,vel} \right] \\
V_y(n) &= \left[ V_y(n - 1) + d_{yn}(n)F_cG_{p,vel} \right] \\
\end{align*}
\]

(14)

A. 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.

B. 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.

Notice that there are other possible fault models such as motor controller or propeller failure, battery failure etc. These faults could be handled by, for example, adding hardware redundancy and additional fault handling mechanisms. The simulator can also be extended with a redundant hardware model and additional fault handling mechanisms. Due to time constraints we did not consider these faults in our simulations and experiments, however, doing so is possible future work.

III. Quadcopter Simulator

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

The CopterModel block runs the same source code for position and velocity estimation, shown in Equation 5 and 6, as the implementation on the hardware quadcopters. The angles \([\theta_r, \theta_p, \theta_y]^T\) are updated from the movement command with a similar response to that of the actual hardware, and the golden run (the true position and velocity state) is updated based on these angles. In addition to the true position state for each copter, CopterModel also updates the perceived position for them. For the perceived position, we have added FaultCheck probes to the various state variables, as described in Section II-B, 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 II. 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 II-A, based on the LDM. Since the ITS-station operates on the perceived position of all copters, it is important that the position-estimation algorithm performs well when there are faults present and that safety margins are large enough to cope with a slightly inaccurate perceived position.

A. 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. This works with up to around 100 quadcopters on the same computer in real time. Notice that the visualization can be omitted and that there are no real time requirements while we are automatically generating and running test sequences using FaultCheck and QuickCheck, so then we can run much larger simulations if necessary as long as the computer has enough memory and the test can finish in a reasonable amount of time.

B. CopterSimGUI

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

Fig. 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.

IV. TESTING COPTER SIM WITH OUR TESTING PLATFORM

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

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

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:

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

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.

B. 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:

```c
// Inject ranging fault
const QString fcStr = QString().sprintf("RangeId%dAnch%d", mItsStation.getId(), anch_int.id);
quickcheck_injectFaultDouble(fcStr.toInt(), Local8Bit().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.

V. 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 III). 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.

A. 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.

VI. 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 straightforward. It gave 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.

VII. 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


