Testing Abstractions for Cyber-Physical Control Systems

CLAUDIO MANDRIOLI[*], Univeristy of Luxembourg, Luxembourg
MAX NYBERG CARLSSON, Lund Univeristy, Sweden
MARTINA MAGGIO, Saarland Univeristy, Germany

Control systems are ubiquitous and often at the core of Cyber-Physical Systems, like cars and aeroplanes. They are implemented as embedded software, that interacts in closed loop with the physical world through sensors and actuators. As a consequence, the software cannot just be tested in isolation. To close the loop in a testing environment and root causing failure generated by different parts of the system, executable models are used to abstract specific components. Different testing setups can be implemented by abstracting different elements: the most common ones are model-in-the-loop, software-in-the-loop, hardware-in-the-loop, and real-physics-in-the-loop. In this paper, we discuss the properties of these setups and the types of faults they can expose. We develop a comprehensive case study using the Crazyflie, a drone whose software and hardware are open-source. We implement all the most common testing setups and ensure the consistent injection of faults in each of them. We inject faults in the control system and we compare with the nominal performance of the non-faulty software. Our results show the specific capabilities of the different setups in exposing faults. Contrary to intuition and previous literature, we show that the setups do not belong to a strict hierarchy and they are best designed to maximize the differences across them rather than to be as close as possible to reality.

CCS Concepts: • **Software and its engineering → Empirical software validation**; **Software testing and debugging**.

Additional Key Words and Phrases: Cyber-Physical Systems, Software Testing, X-in-the-loop Testing

1 INTRODUCTION

Control is at the core of many Cyber-Physical Systems (CPS) and pervasive in modern life [53]. Control is found in many devices, from small consumer electronics like phones, to cars, or space vehicles [10, 44, 47]. Historically, control systems were built using mechanical devices, like hydraulic circuits [42]. Nowadays, they are built using software that interacts with the physical world through sensors and actuators [9]. The ubiquity and criticality of software-based control systems makes their verification and validation of primary importance [15, 67].

A controller comprises of sensors, actuators, hardware and software, and is used to make a physical system behave according to given requirements [9]. The union of controller and physical component is called "control system". Prominent examples of control systems are the cruise control of a car or the control of electric motors like a DC servo (a continuous current motor). In the cruise control, the objective is to ensure that the car reaches and maintains the desired velocity [56]. To achieve this, the control software iteratively reads the encoders attached to the wheels, computes a control action, and actuates it by opening the throttle or pushing the brakes. In the DC servo, the objective is to move the motor shaft to a desired position that varies over time. To reach the desired shaft position, the control software iteratively measures its actual position and applies a voltage to the

[*] This work was done when the author was a PhD student at Lund University, Sweden.

Authors' addresses: Claudio Mandrioli, claudio.mandrioli@uni.lu, Univeristy of Luxembourg, Luxembourg; Max Nyberg Carlsson, max.nyberg_carlsson@control.lth.se, Lund Univeristy, Sweden; Martina Maggio, maggio@cs.uni-saarland.de, Saarland Univeristy, Germany.

1049-331X/2023/8-ART $15.00
https://doi.org/10.1145/3617170

motor to make it move. In both examples, the iteration of sensing and actuation creates a *closed loop* between the physical component and the software. The two parts (controller and physical system) are hence coupled and cannot be evaluated separately.

In control systems, the software plays a crucial role of decision-making. Depending on the application, if this process is incorrect there can be dramatic consequences. Furthermore, modern applications include high levels of digitalisation and integration. For example, the software of a car executes several control systems in parallel (traction control, stability control, anti-lock braking system), also together with the infotainment systems [18, 25]. This makes control software complex, and prone to errors. Unsurprisingly, control software requires a long and costly verification and validation process [23].

During the verification and validation process, engineers spend most time on testing [13, 23, 73]. The main difficulty in testing a control system implementation arises from the necessity of executing the system in a closed loop. Unit testing of the individual components is clearly important, but of limited effectiveness, and system testing is crucial [2, 45]. Given the tight coupling of components, it can be very difficult to identify a fault location. In fact, even when only one component is faulty, the malfunction spreads to all the components in the loop. Furthermore, the physics makes the execution of tests non-deterministic and costly both in time and resources.

To work around the tight coupling of the system and reduce the cost of executing system tests, it is common practice to *abstract* specific components and substitute them with executable models [37]. Furthermore, the use of given components' executable models in the place of their final implementation can enable the system verification earlier in the development cycle, according to the availability of the different artefacts. The choice of which components to abstract defines different testing setups [17, 32, 72]. Said setups are called *X-in-the-loop*, where "X" (e.g., software or hardware) describes which components are included as their final implementation and which components are abstracted. To the best of the authors' knowledge, despite being common industrial practice, the differences in fault-finding capabilities among X-in-the-loop setups have never been studied.

In particular, previous research started from the — often implicit — assumption that there exists a hierarchy among the testing setups. This hierarchy is supposed to manifest itself in terms of the testing capabilities and the coverage achieved with one or another setup. To mention some examples: Zander et al. [72, pp. 13 – 14] and Perez et al. [39, pp. 2] discuss of how each testing setup adds detail to the testing representativeness, Bringmann et al. [17, pp. 3] discusses the increasing level of integration of the different testing setups, Bringmann et al. [16] and Peleska et al. [58] discuss the re-use of test cases across testing setups, and their incremental nature in approximating the real-world behaviour. Accordingly, previous literature uses the naming "*testing levels*" for the different setups, hence implying an ordering. A likely explanation of why this assumption has not been challenged, is that research on the topic is also limited by the development effort required by the implementation of the different setups.

With the aim of filling this gap in the study of the setups differences and of enabling further research, this paper provides the following contributions:

(i) a general discussion of the testing abstractions in control systems' testing (Section 3),
(ii) the development of four complete testing setups for a fully open-source case study [1], that enable the consistent injection of different types of faults (Section 4), and
(iii) the comparison and discussion of said setups in terms of their ability to detect different types of software faults (Section 6).

We address the latter point by answering the following research questions:

RQ1: What are the differences between the testing abstractions with respect to their fault revealing capability?
RQ2: When and why is it beneficial to have different testing setups? What are the principles to be followed when designing the testing setups?
RQ3: What are the domain-specific characteristics of system testing for closed-loop control software?

Our findings confute the common assumption of hierarchy among the setups. We evidence the strengths and weaknesses in fault-finding capabilities of each setup in the verification of functional properties, timing properties, and code (statement) coverage. We provide insights in the best practices to be followed when designing the setups: more specifically, we highlight that the difference in the testing abstractions among the setups is more relevant than the accuracy of each of them. While the results are based on a single case study, the algorithms used — Kalman Filtering and Proportional Integral and Derivative (PID) Control — are the most common choice in control systems. According to an industrial survey [20], 97% of controllers worldwide are PIDs. As a further element of general validity, we note that all control systems share significant commonalities in the implementation structure. This is because they *all* implement an iterated loop of sensing, state estimation, control computation and actuation. Such considerations support the general validity of the case study.

Paper Outline. The paper is organised as follows. Section 2 provides the background on the development of control systems and defines the testing problem addressed in the paper. Section 3 defines the testing setups according to their corresponding testing abstractions. Section 4 presents the implementation of our open-source case study and the choice of the injected faults. Section 5 presents the results of the test flights with the injected faults. Section 6 uses our testing results for answering the research questions and discusses their generalisability and limitations. Section 7 and Section 8 conclude the paper, presenting related work and conclusions.

2 CONTROL SYSTEMS DEVELOPMENT BACKGROUND AND PROBLEM STATEMENT

Control systems regulate physical quantities so that they behave as desired [33]. In practice, control software samples in real time a vector of measurements $y(t)$ from a physical component. Control algorithms use this information to compute the values of a vector $u(t)$ of actuation commands. The actuators affect the state vector $x(t)$ of the physical component. The state vector is linked to the vector $y(t)$ of measurements, creating a closed loop between the physical component and the control algorithm. The control objective is that the state $x(t)$ follows a vector of corresponding reference signals $r(t)$.

The synthesis of a control system starts with the definition of the *control requirements* [34], i.e. the description of how $x(t)$ is expected to follow $r(t)$. The most basic and common control requirements are: (**i**) stability (the system eventually converges to an equilibrium point), (**ii**) set-point tracking (constraining the difference between $x(t)$ and $r(t)$), and (**iii**) settling time (constraining the time needed for $x(t)$ and $r(t)$ to be sufficiently close).

As a practical example, we cast these requirements into a concrete control system: DC servo control. In DC servo control, we want to regulate the motor shaft position, $p(t)$, measured in degrees *deg*. Stability requires that the shaft position eventually reaches a constant value, formally defined as $\lim_{t\to\infty} p(t) = \bar{p}$ where $\bar{p}$ is finite. Set-point tracking may be specified as: the absolute value of the difference between $\bar{p}$ and $r(t)$ is not greater than 2 *deg*, $|\bar{p} - r(t)| \leq 2\,deg$. The settling time requirement may impose a maximum of 10 *s* to bring motor shaft position from 50 *deg* to 70 *deg*: given $v(t_0) = 50\,deg$ and $r(t_x) = 70\,deg$ for $t_x \in (t_0, t]$, then $v(t) \in [68\,deg, 72\,deg]\ \forall t \geq t_0 + 10\,s$.

The next step in the development is the synthesis of a physical component model — usually a set of nonlinear differential equations. For the DC servo, these equations describe the shaft movements in response to the voltage changes. Models are derived using first principles (from physics), data-driven methods, or a combination of the two [8].

Given the requirements and the model, the control engineer, together with application-domain-specific engineers (e.g., electrical engineers in the case of the DC servo), chooses which quantities must be measured and actuated — i.e. the sensors and actuators that have to be installed on the physical component. This defines the measurements and actuation signals available to the control algorithm. In the DC servo example, possible choices are: an encoder mounted on the shaft to measure its angle, and the voltage applied to the main electrical circuit that generates the torque.

Given model, sensors and actuators, control theory provides different classes of algorithms and design methodologies to synthesise a controller that fulfils *a priori* the specified requirements [34]. Examples of such algorithms and methodologies are: PID controllers or state-feedback, and frequency-domain design or Linear-Quadratic Regulator control.

Independently of the application, the vast majority of control design methods specify the controller as a set of differential (in continuous-time) or difference (in discrete-time) equations. To handle discrete inputs, like user commands and operation mode switches, this equation-based controller is complemented with a high-level discrete-state controller, usually specified as a state-machine [31, 52]. Continuing with the DC servo example, the low-level equation-based controller is responsible for using the encoders measurements to actuate the voltage. The high-level discrete-state controller instead handles the control engagement and disengagement and other discrete inputs, e.g., user commands. Hence, the complete controller specification is a combination of a state-machine and equations.

State-machines and differential equations are ideal mathematical objects: their implementation on discrete computers requires approximation. For example discretisation of continuous equations and practical definition of transition signals. Finally, the code of the controller is implemented and executed on hardware, closing the loop around the physical component. In our example, the DC servo control algorithm is translated into code, compiled and flashed onto the motor microcontroller.

The correctness of control software implementations and the satisfaction of requirements depends not only on the code, but also on all the other components in the loop. For example, the resolutions of digital to analog and analog to digital converters, the sensors' noise, and the actuators' performance play a fundamental role in the achieved performance. Different errors in the development can affect the satisfaction of the requirements. After the requirements definition, there could be errors like: (**i**) using modelling assumptions that are not consistent with the physical component or are not detailed enough for the control problem, (**ii**) faulty design of the controller, either in the choice of control type or of control parameters, (**iii**) software faults, and (**iv**) issues with the physical component.

When failures appear in the control system, *the co-dependant implementation and the interdependence of the different components make it difficult to single out the specific fault that is causing the problem* [11, 55]. To identify the source of the problems, engineers use different testing setups that abstract different system components. In this way, they can expose the responsibility of the different parts.

At this stage, the **testing objective** is to verify that the implemented system fulfils the control requirements stated at the beginning of the control system design process. This is done by feeding the system with a sequence of reference values $r(t)$ (the **test inputs**) and evaluating the requirements over the output traces $y(t)$. The most common practice for verifying control properties is to look at **step responses**, as those allow the direct verification of stability, tracking and settling time properties [8]. We show how this is common control engineering practice with an example. Furthermore, we use the example to showcase the difficulty of root causing failures in control systems implementations.

Motivating Example. We consider the implementation of the control of a DC servo motor where the objective is to move the rotating motor to reach a desired angle.[1] In our example, the reference r is the desired angle, the output y is the measured angle, and the control action u is the voltage fed to the motor. We use the Simulink[2] simulation environment for simulating the motor's physics, the encoder's quantization, and the pulse-width-modulation that implements the digital to analog conversion. For the implementation of the control algorithm, we leverage the possibility of incorporating custom C code in the Simulink environment and consider a fixed-point implementation of the controller.

[1] Such systems are also commonly known as servo systems.
[2] https://www.mathworks.com/products/simulink.html

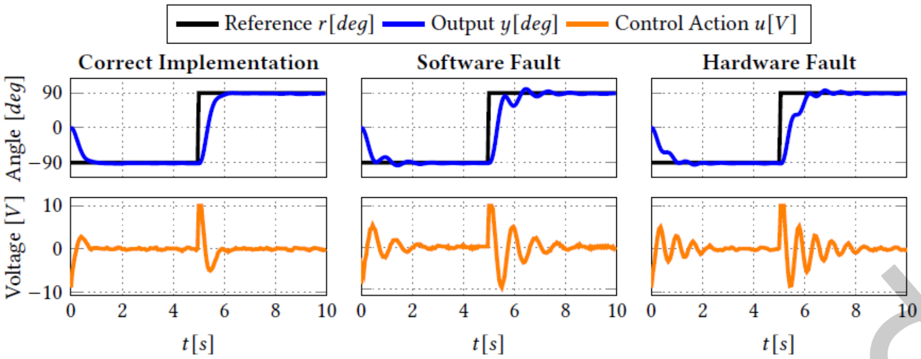
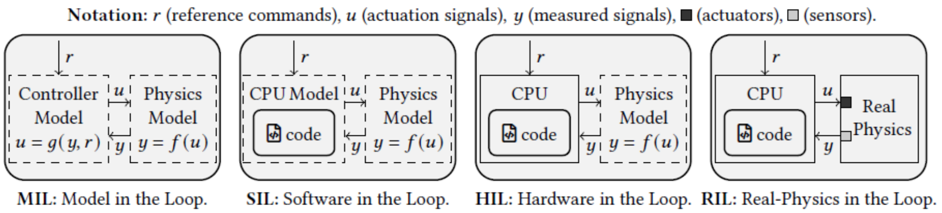


Fig. 1. Step responses of the DC servo motor of the motivating example. The plots show how the control system makes the angular position of the motor (the blue line) follow the reference (the black line). The orange line shows the voltage actuated to the motor. The plot on the left-hand side shows the system performance with the correct implementation of the control system. The central plot shows the system performance in presence of a software fault in a matrix-vector multiplication. The plot on the right-hand side shows the system performance in presence of a hardware fault that causes incorrect timing of the software execution.

We perform three different step response tests: one with the correct system implementation and two with different types of faults. The first injected fault is an incorrect implementation of the state estimator of the controller: more specifically, a matrix-vector multiplication is altered. This emulates an error by the *software developer* at the time of implementing the control algorithm as C code. The second injected fault is incorrect timing of the control algorithm execution: more specifically, the software is executed every $0.06[s]$ instead of $0.05[s]$. This emulates a fault introduced by the *hardware engineer* when designing or configuring the hardware platform. We show the results of the three tests in Figure 1. In the plots, the black lines represent the reference values that we ask the physical system to follow. The controller then performs a sequence of steps for the physical quantities to meet their reference values. The blue lines show the actual angular position of the motor, i.e., the quantity that we are trying to control. The orange line in the lower plots are the voltages fed to the motor, i.e., the control signal.

Step responses allow to directly verify the main requirements of the control algorithm [8]. We observe that the system is stable in each test, meaning that the blue and orange lines do not diverge. Also, the controller achieves *reference tracking*, as the output eventually (i.e., after a transient phase) converges to its reference value (blue and black lines). We also observe that it takes approximately one second (in the test with the correct implementation) for the blue line to reach the black line after a step change. This is the settling time and measures the (reaction) speed of the control system. Finally, we observe that the faulty tests show significant oscillations after the step changes in the reference. This is apparently undesirable behaviour and exposes the presence of a fault.

The oscillations in the two faulty tests are similar and there is no way to state, on the sole base of these tests, which is the root cause of the faulty behaviour: i.e. the software fault or the hardware fault. However, an extra test that includes the software implementation of the control algorithm, but not the hardware, would enable the distinction between the two scenarios. If the test does not show the oscillations, then the fault has to be in the hardware, because the hardware is *abstracted* in the testing setup. Differently, if the extra test also shows the oscillations, then we would be able to exclude the hardware from the possible causes and conclude that the fault is in the software. This conclusion can be drawn because the hardware is *abstracted* in one of the setups. Said in


Notation: r (reference commands), u (actuation signals), y (measured signals), ■ (actuators), □ (sensors).
r
Controller Model
$u = g(y, r)$
u
y
Physics Model
$y = f(u)$
CPU Model
code
CPU
Real Physics

MIL: Model in the Loop. SIL: Software in the Loop. HIL: Hardware in the Loop. RIL: Real-Physics in the Loop.

Fig. 2. Testing setups for different testing levels. Dashed lines indicate that the corresponding component is simulated, while solid lines denote the component execution. Notation: c (external commands), u (actuation signals), y (measured signals), ■ (actuators), □ (sensors).

other words, such extra test would show the undesired oscillations only if the fault is in the software but not if it is in the hardware, thus enabling the distinction of the responsibility for the faulty behaviour.

This example shows two things: first, it shows how the step response can be used to verify the main control requirements. Second, it shows that the use of different testing setups helps root causing of failures in control systems. However, this is possible only if there is a thorough understanding of what the different abstractions of the different setups are. Furthermore, implementing a setup, and executing the corresponding tests, comes with costs, hence the choice of how many and which setups to consider in a given application should be optimised. In this paper we set out to investigate what are the different abstractions involved in the common testing setups used in control systems and evaluate what implications they have in the fault-finding process. This will help engineers in the design of their testing infrastructure for CPSs. In the next section, we define and describe the most common testing setups for control systems and the related design choices that practitioners have to make [17, 32].

3 TESTING ABSTRACTIONS

Section 2 showed the multi disciplinary nature of control systems. As a consequence, *system-level testing* is of fundamental importance. Like shown in our motivating example, it allows the engineers to establish the different responsibilities during the development process. Accordingly, it is one of the main activities software engineers perform in this context [23].

The overall structure of a CPS control system is usually represented with a block diagram similar to the ones shown in Figure 2. A cyber *controller* block is connected to a *physical component* block to form the closed loop. The system has three main components: (**i**) the physical component, (**ii**) the software implementing the control algorithm, and (**iii**) the hardware executing the software. The interaction between the controller and the physical component happens through actuators and sensors. The controller can also receive inputs from other software components or from human operators. In the DC servo control example, the hardware is a control unit (usually a microcontroller) and the software is the code executed by the control unit, implementing the control algorithm. The external inputs are the commands received from the driver (e.g., engagement and disengagement commands, and the desired shaft angle).

Potentially, components can be **abstracted** — i.e. substituted with simulation models — so that the other components can be tested in isolation. Abstracting one or more components defines a testing setup [72]. When a component is abstracted, it is important that its simulation model and interaction with the other components are representative of the actual implementation. Said in other words, *for an abstracted system-level testing setup to be effective, there are associated assumptions that have to hold: these assumptions concern the validity of the models, their implementation, and their interaction.* Figure 2 provides a graphical representation of the closed loop for

Table 1. Abstractions for system-level testing of control systems. Comparison among: Model in the Loop (MIL), Software in the Loop (SIL), Hardware in the Loop (HIL) and Real-Physics in the Loop (RIL). The setup comprises controller code (C), hardware (H) and real physical component (R); 🖩 indicates that a component is simulated, 🔌 that its real instance is used.

Name	**Setup**			**Underlying Assumptions**
	C	H	R	
MIL	🖩	🖩	🖩	(**i**) physical component model is accurate, (**ii**) controller model corresponds to implementation
SIL	🔌	🖩	🖩	(**i**) physical component model is accurate, (**ii**) hardware model captures the relevant properties (e.g., timing and instruction set)
HIL	🔌	🔌	🖩	(**i**) physical component model is accurate, (**ii**) execution of input/output hardware peripherals is not affected
RIL	🔌	🔌	🔌	–

each testing setup: the dashed blocks are emulated and solid ones are implemented. Table 1 summarises the main testing setups and their fundamental assumptions.

To abstract a component, a corresponding executable model has to be provided. The control synthesis phase produces executable models for both the control law and the physical component. On top of these, in this paper we consider leveraging a hardware emulator.[3] In the following sections we discuss possible implementation choices for each setup and the consequent testing abstractions — i.e. the set of assumptions that have to hold for the testing setup to be effective in detecting faults.

3.1 Model in the Loop (MIL)

At the model-in-the-loop abstraction level, all components of the system are simulated through models, as shown in Figure 2-MIL. The execution of said models requires a dedicated simulation environment. During the system development, MIL testing is performed in two ways: during the control design, and as part of the system testing (so called model-testing [15]). For control design, the control engineer develops an executable model of the physical component, represented in Figure 2-MIL by the function f, and a control law g, that uses measurement signals and control commands to compute the actuation signals. The models are defined as differential equations, difference equations, and state machines [34], they can be implemented using common simulation software, like MATLAB[4] or Modelica.[5] In this way, the closed loop is tested to verify that the algorithm meets the expected performances and can be used to fine-tune the parameters [69].

Testing Abstractions. MIL testing is completely simulation-based, and hence it fully relies on modelling assumptions. These can be divided in two categories depending on what they concern:

(**i**) Physical component-related assumptions that concern the differences between the model of the physical component and its actual physical realization. More specifically, such assumptions relate to the physical model being an adequate representation of the physical component.

(**ii**) Controller-related assumptions that concern the differences between the control algorithm model and its implementation. More specifically, such assumptions relate to the software and hardware being a consistent implementation of the control algorithm.

[3] Apparently, such emulator is not always available and requires a development effort. This has to be considered during the design of the testing process and infrastructure.

[4] https://www.mathworks.com/products/matlab.html

[5] https://modelica.org/modelicalanguage.html

Examples of assumptions on the physical component are: neglected dynamics (like friction in the shaft movement), modelling approximations (like linearisation of nonlinear models), and neglected phenomena (like friction and road surface variability). Control theory provides metrics and rules-of-thumb to quantify the *robustness* of a control algorithm to non-ideal behaviour. However, these metrics also rely on assumptions, and hence need verification.

Examples of controller-related assumptions are: ideal timing (instantaneous execution) and infinite numerical precision. Moreover, not necessarily all of the required control features are implemented at this level. For example, a controller with different modes of operation may benefit (in terms of simplicity) from these modes being implemented and verified individually, neglecting the mode switching. The DC servo example, might include controllers that set the shaft position or controllers that instead make the shaft rotate at a desired speed. If this is the case, the mode switching code has to be tested in other setups.

3.2 Software in the Loop (SIL)

In the software-in-the-loop setup (Figure 2-SIL), we include the actual software implementation, while hardware and physical component are still abstracted. The physical component is implemented using models that are similar to (or the same as) the ones used for MIL testing.[6] On the hardware side, different choices are viable, from simulating only a few hardware components — like for example in our motivating example where we emulated the encoder and the pulse width modulation — to complete cycle-accurate hardware emulation. A simple alternative is to test the code in a general purpose machine. The code is then compiled for and executed on a machine different from the target one, hence abstracting the hardware and the execution environment. Under the associated assumptions, this enables testing of the functional component of the software, i.e., if the control law g is implemented correctly. However, other non-functional properties (e.g. execution time) cannot be verified, since they relate to system components that are abstracted. A more detailed alternative is hardware emulation: tools like gem5[7] and Renode,[8] can provide a higher degree of testing significance. Such a solution is often preferred in embedded systems (hence in control systems as well) given the strong coupling between hardware and software. In this way, the software is compiled for the target hardware. Among other things, hardware emulation enables the testing of the interaction with the Real-Time Operating System (**RTOS**) and possibly low-level software routines that interface with the sensor and actuator peripherals [6].

Testing Abstractions. In SIL, the testing abstractions can still be divided into two sets: the first set is equivalent to MIL and relates to the physical component modelling, that needs to be accurate. The second set of abstractions is related to the environment in which the software is executed, varying significantly according to the specific choices made for the hardware abstraction. In general, these require that execution environment is representative of the actual one. Such abstractions mainly include:

(i) software environment (meaning the interaction with other software components: for example the RTOS, if the code is executed on machine other than the target one),
(ii) hardware (e.g. support for floating point arithmetic [35]),
(iii) time modelling (the timing of the software execution has to be consistent with the physics simulation and possibly with other events, like user commands), and
(iv) input and output definitions (measurement and actuation signals are representative of the real ones, e.g. with respect to measurement units).

[6] Here, models might need refinement. In the DC servo example, during the control design process, the engineer might assume direct control over the shaft acceleration. In the SIL setup, on the contrary, the simulation needs to include the fact that the actuation signal is the voltage command sent to a digital-to-analog converter that uses pulse-width modulation to generate an analog voltage signal.

[7] http://www.gem5.org

[8] https://renode.io

3.3 Hardware in the Loop (HIL)

The hardware-in-the-loop setup includes the target hardware in the testing process, as shown in Figure 2-HIL. The control software is now executed on the target computing platform — e.g. the microcontroller DC servo example — and the model of the physical component is simulated on a different machine. The actuation signals produced by the software are extracted and fed to the physics simulator, while synthetic sensor readings from the simulator are fed to the hardware. The main design choices for this setup concern (**i**) the level at which the measurements and actuation signals are redirected, (**ii**) and the synchronisation between the controller execution and the physical component. For the first item, options range from using a debug port and accessing the memory registers of interest, to manipulating the software so that it interacts with the simulator instead of the actual peripherals. If signals are intercepted at lower levels, more details will be required for the model simulation: for example, in the DC servo example, position readings might have to be scaled to the encoder resolution instead of being in the physical units of measure. As an alternative, dedicated testing hardware can be developed so that it interfaces with the simulator at the physical connection level (i.e. I/O pins) instead of requiring that the software is redirected. This allows better coverage of the low-level firmware. Concerning the time synchronisation, the testing setup must ensure the consistency of time between the target hardware and the simulated physics; this can be done by performing the physics simulation and the I/O operations in real time. Such a solution is however difficult to realise [33] and explicit synchronisation points might be needed — e.g. every millisecond the hardware is halted, then outputs are read, the physics is simulated, and sensor values are written before execution is resumed.

Testing Abstractions. Apparently, also the HIL setup includes the abstractions associated to the modelling of the physical component. The two sets of design choices mentioned above are associated to respective testing abstractions. Intercepting the actuation and sensor signals at a higher level will possibly exclude more of the software that handles said signals in the control system. Consequently, this software is abstracted from the testing and assumptions have to be made about its behaviour. Analogously, the chosen synchronisation mechanism (if a real-time simulation is not implemented) can abstract timing phenomena from the test. For example, if the controller and physical simulator are synchronised every millisecond, events that happen at a higher rate are abstracted. To summarize, the HIL testing abstractions concern:

(**i**) the input-output interactions of the hardware with the physical world, and

(**ii**) the consistent evolution of computational time in the hardware and the evolution of time in the physical component.

3.4 Real-Physics in the Loop (RIL)

In the RIL setup the actual physical component is included in the closed loop, therefore the full implementation of the CPS can be used and there are no testing abstractions. Extra sensors could be installed on the physical component and prototypes might be used in place of production models: such solutions are highly application-dependant and therefore excluded from this discussion. Once this aspect is excluded, the RIL testing effectiveness is mostly dependant on the choice of the test cases and their evaluation (which is apparently also important in the other setups). However, in this work we focus on the design of the setups rather than of the test case generation and evaluation.

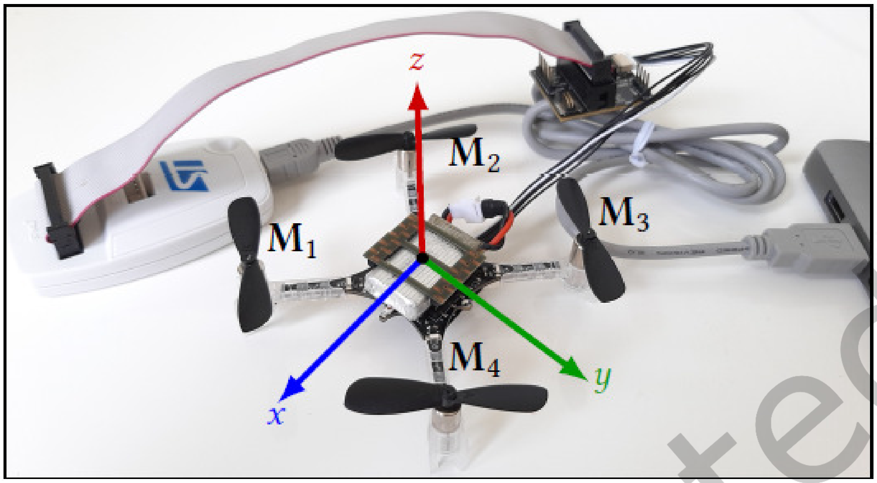


Fig. 3. Crazyflie 2.1 with the STM debugger link.

4 EXPERIMENTAL SETUP

In this section, we present our case study to empirically investigate the differences between the testing setups described in Section 3. We developed and implemented the MIL, SIL, HIL and RIL testing setups for the Crazyflie 2.1 quadcopter[9], developed by Bitcraze[10] shown in Figure 3.

We developed the setups with the objective of allowing *the consistent injection of the software faults across the setups*. We did so by allowing only modifications strictly necessary to the setup implementation in the drone software when implementing the different setups. We provide a detailed report on the modifications and the setup design choices behind them. This is crucial to avoid biases in the study caused by the differences in the fault implementations and to assess the general validity of the results.

With the different setups, we first run the control software. We then inject faults in it, and run tests in each abstraction configuration. We inject faults in the implementation of the control software, hence in the SIL, HIL and RIL setups. The MIL setup abstracts the software implementation of the control algorithm and therefore software faults cannot appear in it. For debugging purposes (e.g., to reproduce a failure in a simulated environment), practitioners might still want to reproduce some software faults in the MIL setup. Since it was possible to only reproduce four of the ten faults used in this paper, we did not consider them sufficient to compare the fault-revealing capabilities of this setup with the others. Hence, we focus the experimental campaign on the SIL, HIL and RIL setups.[11] We finally compare both quantitatively and qualitatively the tests results to evaluate the different fault-revealing capabilities of the setups.

This section is organised as follows. In Subsection 4.1 we motivate the choice of the Crazyflie and provide the relevant background information.[12] In Subsection 4.2 we describe our implementation of the testing setups. We

[9] https://www.bitcraze.io/products/crazyflie-2-1/

[10] https://www.bitcraze.io/

[11] In the associated repository, we provide test results for the faults that we were able to reproduce in the MIL setup: https://github.com/ManCla/testing-abstractions/tree/main/testing-frameworks/mitl/pdf.

[12] For the sake of reproducibility, in this work we refer to the Crazyflie software at commit `23e9b80c` available at https://github.com/bitcraze/crazyflie-firmware/commit/23e9b80caa9137d2953ae6dce57507fda1b05a8c.

then describe the results of test flights with nominal software in Subsection 4.3, and finally we discuss choices for the faults to inject in the software in Subsection 4.4. A repository accompanies the submission [1], providing the code we developed for the testing setups, documentation to reproduce all of the tests (including the ones that require the Crazyflie hardware, i.e., HIL and RIL), but also pre-recorded flight data, and detailed plots for each of the tests.

4.1 Crazyflie Quadcopter

The choice of the Crazyflie case study is motivated by two main reasons. First, the control system of the quadcopter is both not trivial and based on the most used control algorithms, making it a practically relevant case study. In fact, the Crazyflie is known to the research community; it is used for both education and research, e.g., quadcopter control design [19], swarm robotics [5, 30, 46], distributed [68] and robust control [51]. Second, both the Crazyflie software[13] and hardware[14] are completely open-source. We therefore have complete knowledge about the design of the system, which allows us to build a testing infrastructure for all the MIL, SIL, HIL, and RIL setups. In particular, using the open-source hardware specification, we can build the hardware emulator for the SIL testing. The use of the hardware emulator allows to run the same binaries in the different setups, hence ensuring the consistent injection of faults in each of them. Similarly, we use the open-source code for both SIL and HIL testing. To ensure reproducibility of the results and make the artefact available to the research community for further investigation we used only open-source tools for the implementation of the infrastructure needed in the different setups.

In this work, we consider a Crazyflie equipped with an Inertial Measurement Unit (**IMU**) sensor, a camera for optical flow, and a vertical laser ranging sensor.[15] The vertical laser provides a direct measure of the distance from the ground, while the combination of optical flow and IMU data allows the drone to estimate the horizontal speed. Such setup is very common in drones, a notable example being the NASA Ingenuity drone flying on Mars [60]. Furthermore, this setup does not require external measurements systems (like the lighthouse positioning system[16]), making it more portable.

As discussed in Section 2, the controller of the drone is constituted of two main components: one high-level discrete controller and one lower-level continuous controller. The high-level discrete controller[17] mainly handles the external user inputs and generates reference signals $r(t)$ for the low-level controller. The low-level controller combines a state estimator and a feedback controller.

In the repository associated to the submission we provide a detailed report on the control design of the Crazyflie.[18] Here, we limit ourselves to the discussion of the main quantities involved, since those are needed for interpreting the results of the tests. The state estimator uses sensor readings to estimate the state $x(t)$ of the drone in real time, producing the estimated value $\hat{x}(t)$. The state vector $x(t) = [p(t), v(t), q(t), \omega(t)] \in \mathbb{R}^{13}$ includes the drone position $p(t) \in \mathbb{R}^3$, the drone velocity $v(t) \in \mathbb{R}^3$, the attitude $q(t) \in \mathbb{R}^4$ and the attitude rate $\omega(t) \in \mathbb{R}^3$. Figure 3 shows the axes definition for p. The attitude q is expressed using quaternions[19] and encodes the three angles: pitch (rotation θ around y axis), roll (rotation ϕ around the x axis), and yaw (rotation ψ around the z axis). The feedback controller uses the estimated state $\hat{x}(t)$ together with the reference values $r(t)$, to compute the voltage signals to be issued to the motors M_1, M_2, M_3 and M_4 illustrated in Figure 3.

[13] https://github.com/bitcraze/crazyflie-firmware

[14] https://github.com/bitcraze/hardware

[15] https://www.bitcraze.io/products/flow-deck-v2/

[16] https://www.bitcraze.io/documentation/system/positioning/ligthouse-positioning-system/

[17] https://www.bitcraze.io/2020/05/the-commander-framework/

[18] https://github.com/ManCla/testing-abstractions/blob/main/Technical_Report.pdf

[19] Quaternions are a four-dimensional extension of complex numbers, and a very convenient tool to represent rotations in the three dimensional space.

When flying with optical flow data, the state estimator is implemented as an Extended Kalman Filter (**EKF**) [49, 50], while the feedback controller is a set of cascaded PID controllers [7]. The setup with state estimator and feedback controller is standard in control theory and found in most control systems. The control design process provides equations used to model the quadcopter and equations to describe the estimator and the controller [22, 26]. Such equations can be found in our technical report.

4.2 Crazyflie Testing Setups

This section discusses our implementation of the testing setups for the Crazyflie. In every setup, we use the same simulator of the physical component, so that the tests expose only differences in the control algorithm executions and the associated testing abstractions.[20] The testing setups of SIL, HIL and RIL require changes in the Crazyflie software, for which we provide patch files and application instructions [1].

MIL. We implemented in Python a physical model to describe the Crazyflie and its controller [26]. We use the SciPy module[21] to integrate the differential equations describing the physics. In MIL, several aspects are abstracted with respect to the software implementation of the controller. Some examples are: (**i**) the computation of the matrix exponential is performed using the NumPy[22] linear-algebra library, while, in the real Crazyflie software, the calculation is approximated, (**ii**) each floating point variable has double precision, while the firmware uses single-word floats, and (**iii**) our model implementation is single-threaded, while in the software the algorithm is distributed over different threads. The physics model is based on first principles, however it also abstracts different phenomena. Some examples are: (**i**) the flexibility of the structure of the drone is abstracted (hence assumed infinitely rigid), (**ii**) the dynamics of the electric motors is abstracted (the relation between the voltage fed to the motors and the vertical thrust is assumed quadratic), and (**iii**) the differences between the two horizontal axes are abstracted (the drone is assumed symmetric).

SIL. In our SIL setup, we rely on the open-source hardware emulator Renode. Bitcraze maintains its own fork of Renode[23] and of the Renode-Infrastructure[24] which contains the emulators of the hardware peripherals. We implemented the platform emulator, which is able to execute the binaries as they are compiled for the target hardware. We also implemented the infrastructure to allow communication between Renode and our simulator of the physics. Said infrastructure leverages the possibility of exposing, along with a Renode emulation, an OpenOCD[25] interface. OpenOCD is an interface used for debugging of embedded devices. It exposes a Telnet port through which it is possible to read and write to specific memory addresses, or insert breakpoints. Some changes were required in the software to interface with the physic simulator: (**i**) in the Flow deck driver, the low-level interaction with the camera is disabled, (**ii**) in the Z-ranger driver, the low-level interaction with the ranging sensor is disabled, (**iii**) in the motor driver, no output is written to the motors, (**iv**) in the IMU driver, the sensors calibration is skipped, (**v**) in the Kalman filter, a division by zero check has been added. (**vi**) debug variables are added in `mm_flow.c` and `mm_tof.c`. For the interested reader, the exact changes can be found in the patch file. When compiling the code, our changes are triggered by defining the preprocessor macro `SOFTWARE_IN_THE_LOOP`.

The most frequent interaction with the physics is the sampling of the IMU sensors, which happens every 1 *ms*. This periodic event is triggered by the IMU itself which sends an interrupt to the CPU. In our SIL setup, we use a Python script to iteratively: (**i**) simulate the physics for 1 *ms*, (**ii**) feed the synthetic sensor data to the hardware emulator, (**iii**) trigger the sensor interrupt, and (**iv**) run the emulator. We empirically observed that the virtual time in the emulator is dilated. More specifically, the 1 *ms* software tick of the RTOS does not always

[20] Investigating the use of different models for the physics is a very interesting research problem, but it is out of the scope of this work.

[21] https://docs.scipy.org/doc/scipy/tutorial/integrate.html

[22] https://numpy.org/doc/stable/reference/routines.linalg.html

[23] https://github.com/bitcraze/renode/tree/crazyflie

[24] https://github.com/bitcraze/renode-infrastructure/tree/crazyflie

[25] http://openocd.org/

increase when the emulator is issued to run for one millisecond. For this reason, at each iteration our script checks whether the software tick has increased or not and run the emulator until the tick increases. This check suffices to keep the simulated physic time and the RTOS time synchronised, at least to the resolution at which the sensors are sampled. Differences from execution on the real platform can still happen in other tasks that are timed on something else than the RTOS tick.

To summarise, our SIL setup for the Crazyflie is based on the following assumptions and abstractions:

(**i**) the physical model is representative of the physical component and of the sensors,
(**ii**) the emulator of the CPU is accurate,
(**iii**) the synchronisation between the physical model and the emulator is representative of the actual interaction, and
(**iv**) the hardware of the Flow deck is not emulated.

HIL. In our HIL setup, we chose not to use dedicated hardware as it is not necessary to be able to consistently inject bugs across the setups. Hence, we used the normal hardware as it is sold by Bitcraze. To enable low-level access to the hardware, we used the debugger link ST-LINK/V2,[26] also depicted in Figure 3. We used OpenOCD, the interface introduced in the SIL setup, to communicate with the debugger, and communicate with the CPU. We introduced the following changes in the software to interface with the physics simulator: (**i**) in the Flow deck driver, the low-level interaction with the camera for optical flow is disabled, (**ii**) in the Z-ranger driver, the low-level interaction with the laser ranging sensor is disabled, (**iii**) in the motor actuation, no output is written to the motors, (**iv**) the IMU sensor is never read, (**v**) the sensor thread is timed on the RTOS ticks instead of the external IMU interrupt, (**vi**) in the Kalman filter implementation, a check for division by zero has been added. (**vii**) debug variables are added in the files `mm_flow.c` and `mm_tof.c`, (**viii**) two assert statements in `uart_syslink.c` are skippeed.[27] These changes are introduced with the provided patch file and triggered by defining the preprocessor macro `HARDWARE_IN_THE_LOOP`.

To synchronise the hardware with the physics simulator, we issue a breakpoint when the IMU sensor is read. When the breakpoint is hit, our Python script performs the following operations: (**i**) read the motor values, (**ii**) simulate 1 *ms* in the physics, (**iii**) feed the sensor readings to the CPU, and (**iv**) issue the CPU to resume execution.

To summarise, our HIL setup for the Crazyflie is based on the following assumptions and abstractions:

(**i**) the physical model is representative of the physical component and of the sensors,
(**ii**) the synchronisation between the physical model and the emulator is representative of the actual interaction (in a different way compared to the SIL abstraction),
(**iii**) the IMU interrupt is not used, and
(**iv**) the hardware of the IMU sensors and of the Flow deck is not executed in the same way as in normal flight.

RIL. Finally, our RIL testing setup consists of running the Crazyflie with its nominal software. We use the Micro SD card deck to log flight data.[28] When compiling the code, the changes needed for the logging are triggered by defining the preprocessor macro `PROCESS_IN_THE_LOOP`. Our MIL, SIL, and HIL setups are deterministic, meaning that, when executed twice with the same inputs they will generate the same output. Instead, the RIL setup is not deterministic, because of the uncertainties related to the physical part of the system. For this reason we performed 30 test flights in RIL with the nominal software to assess the repeatability of the RIL experiments.

[26] https://www.st.com/en/development-tools/st-link-v2.html
[27] The assert statements are related to the communication with the onboard microcontroller. In HIL they might be triggered and halt the CPU because the breakpoint interferes with the communication.
[28] https://store.bitcraze.io/products/sd-card-deck

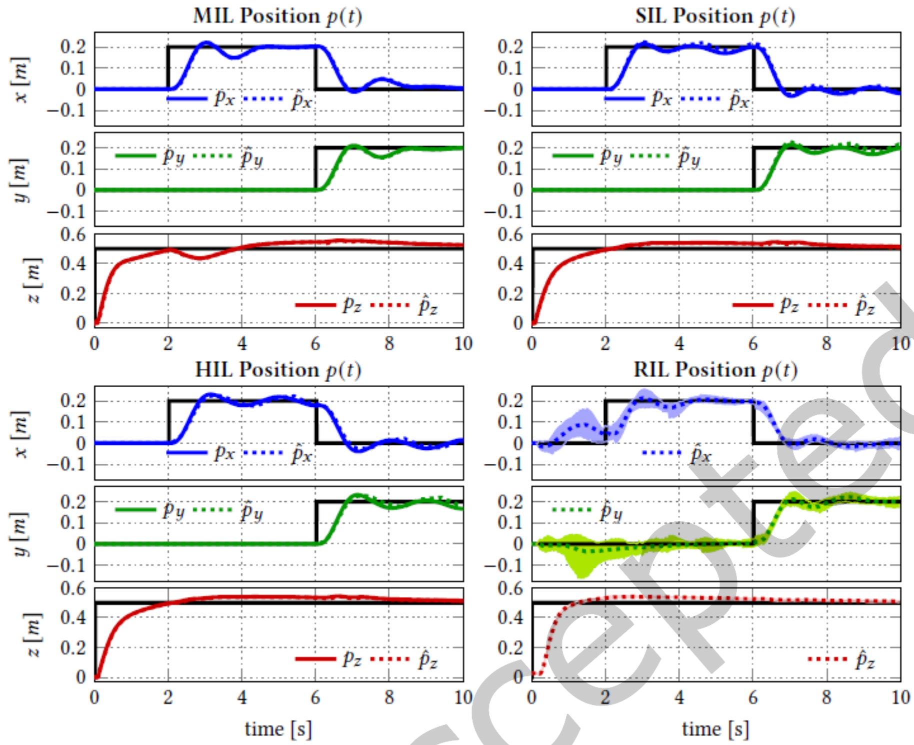


Fig. 4. Nominal flight tests in the MIL, SIL, HIL, and RIL setups. For each axis x, y and z, the solid coloured lines show the drone's true position (when available). The black lines show the step references. The dashed lines show the position as estimated by the drone control software. For the 30 repeated RIL flights, at each time point, the dashed lines show the average over the 26 successful flights of the estimated state. Furthermore, the shades show the area between the maximum and minimum value measured at each time step.

4.3 Nominal Software

We now describe the test flights, using the MIL, SIL, HIL, and RIL setups. Initially, we discuss MIL tests results. We then introduce the implementation of the control software, in its nominal state (as released by Bitcraze).

Figure 4 shows plots of flight with the software as released by Bitcraze in our different setups. The flight sequence consists of a take-off phase from $t = 0$ to $t = 2$ where the drone is given the time to follow the altitude reference step to $r_z = 0.5$. Afterwards, we issue the setpoint step change in the x direction, $r_x(2) = 0.2$, followed at time $t = 6$ by a setpoint step change in both the x and the y directions, $r_x(6) = 0.0$ and $r_y(6) = 0.2$. As mentioned in Section 2, those *step responses* expose the main properties of a control algorithm thanks to their broad frequency spectrum [8]. We chose such step values in order to minimise the likelihood of RIL tests failing just because of sensor noise. In fact, the optical flow from the the camera, and the vertical readings from the laser ranging sensor, deteriorate with higher altitudes and larger movements. Therefore, larger values would introduce tests failing just because of sensor noise circumstantial to that specific execution (instead of being related to the injected faults). Furthermore, a recent paper on the automatic detection of software faults in CPSs showed that the majority (in the case of said paper 80%) of control-related software faults appear in normal operation nor they need specific environmental conditions (and therefore trajectories) to be exposed [66]. Apparently, exhaustive testing of the controller implementation requires more tests, and test case generation for CPSs is an active research topic [72]. In this work, we focus on the differences among the testing setups, rather than how to achieve exhaustive testing.

In the figure, the three top-left plots show the position of the quadcopter in the x, y, z coordinates in the MIL setup.[29] For each plot we include three lines:

- The dark solid line is the desired position r that the drone is expected to reach at every time instant.
- The solid coloured line p is the actual position of the drone from the simulated physics for the MIL, SIL, and HIL setups.
- The dashed coloured line $\hat{p}$ is the position estimated by the drone. As part of the control algorithm, the drone needs to estimate its own position, so that it can take corrective actions through the actuators and reach the desired position. Apparently, when the drone is functioning correctly, the dashed line converges to the continuous line (i.e., the drone correctly estimates its own position).

This test shows that the model of the controller is able to control the model of the physical component. Guarantees on the behaviour of the actual control system are however subject to the validity of both physical component and controller models, and on the implementation details [4] — i.e. the testing abstractions discussed in Section 3.1.

The top-right and bottom-left three plots in Figure 4 show the same test flight respectively in the SIL and HIL setups, using the same conventions. The bottom-right three plots show the results of the repeated tests obtained with the physical component in the RIL setup. In RIL, there is no physics model involved and ground truth is not available, so we only display the position estimated by the quadcopter. Among the 30 RIL flights performed 4 failed without apparent reason, resulting in immediate crash. One possible explanation, as the producers suggest on their website, is that the IMU moving parts can get stuck at times. Using the successful 26 flights, we plot the average over the different flights of the estimated position (dotted lines), and the range between the maximum and minimum estimation. The RIL flights show consistent results, with the exception of the first 2 seconds. At take off, the turbulence caused by the ground effect can make the drone unpredictably oscillate. We also note that the z direction control is more accurate. This is due to the higher performance of the laser sensor compared to the optical flow. Given the general consistency of the RIL tests we did not perform repeated tests for the faulty software, besides the ones to verify that the immediate crashes were not caused by the IMU getting stuck, as discussed above. In fact, performing repeated crashing tests risked damaging the hardware (and could affect the results of the subsequent tests).

While the general behaviour is consistent across the setups, few differences arise. In the SIL, HIL and RIL setups, the drone oscillates around the reference position in the x and y directions: this is due to the optical flow quantisation caused by the camera pixels. Movements smaller than the resolution of the camera are not detected. When the flow reading changes, the controller reacts at once, and the drone oscillates. This quantisation is abstracted in the MIL setup, hence not seen. In the MIL setup, the drone loses some elevation (z position) while performing the step in the x direction. This is caused by the loss of vertical thrust when the drone tilts to move laterally. Our tests show that the software implementation of the controller is robust to this disturbance. Finally, the ground effect is not captured in the physics model hence observed only in the RIL setup. Such phenomenon is chaotic (in the mathematical sense) and difficult to model, hence, it is often neglected in simulated setups.

We note the general consistency across all the setups in nominal conditions. In the next section, we show that when faults are present in the software implementation, the testing setups exhibit significant differences.

[29] More comprehensive plots for all the nominal and faulty test scenarios can be found in the associated repository [1].

4.4 Faults Design

We inject faults in the control software to expose the differences between the testing abstractions and highlight the capacity of each of them to unmask errors in the controller implementation. Unfortunately, it was not possible to mine the Bitcraze repository for faults, as the developers do not use consistent practices to mark issues and commits associated with the control software faults, and frequently squash commits losing part of the version history. Furthermore, to the best of the authors knowledge, there exists no database of faults in control software.

Therefore, for obtaining faults to inject in the software we used two different methods: (**i**) We selected two solved issues in the Bitcraze repository: the faults we used were suggested by Bitcraze engineers, because they struggled to reproduce and identify them. (**ii**) We took faults types from the close research field of faults in robotics systems: specifically, we considered the works of Wienke et al. [71] and Steinbauer et al. [62] to retrieve common types of faults and used the descriptions and examples to develop faults to inject. The scopes of the cited works are wider than ours as it relates to the whole robotic system and not just the control system. Hence, we manually filtered fault types that do not relate to the control system implementation — e.g. faults in communication protocols.

Wienke et al [71] use a practitioners survey to identify different categories of faults and provide some example for each category. Said categories are (with an example from the original work):

- algorithms and logic (e.g. erroneous mathematical computations),
- resource leak (e.g. not closing a no longer needed connection),
- skippable computation (e.g. executing the same computation multiple times),
- configuration (e.g. erroneous initialisation of an address),
- threading (e.g. incorrect timing code), and
- communication (e.g. incorrect address in the radio communication stack).

Among said categories we excluded communication, as it apparently does not relate to the implementation of the control system performance. We also exclude resource leak, and skippable computation since they concern the embedded computing performance of the system rather than the control loop. For example, a memory leak is likely not seen in the control system performance, since it should not affect the functional properties of the software. Similarly, a repeated computation is not harmful, as the control software is supposed to be executed in an infinite loop. Such faults can become an issue when affecting the execution timing of the code, timing faults are however included in the threading class.

Table 2. List of injected faults and corresponding test results. The fault names correspond to the patch files and flight plots in the repository [1]. For each fault, we report the corresponding categories covered among the types of faults in robotics faults highlighted in previous literature [62, 71] as well as the number of lines of code changed (LOC). For each fault and setup, we report in the last three columns the Integrated Squared Error (ISE) and whether the test flight was impaired ✈̸ or not ✈. We note naming discrepancies between the two works used for the fault classification: e.g., a missed deadline is considered a threading fault by Wienke et al. [71] and algorithmic by Steinbauer et al. [62]. This does not affect our use of those classifications as we independently want to cover the relevant classes proposed by the two studies.

Fault Name	**Category from Wienke et al. [71]**	**Category from Steinbaueret al. [62]**	**Changed LOC**	ISE-**SIL**	ISE-**HIL**	ISE-**RIL**
nominal software	–	–	0	134 ✈	135 ✈	202 ✈
`voltageCompCast`	–	batteries/low-level drivers	2	2813 ✈̸	2813 ✈̸	2488 ✈̸
`initialPos`	configuration	algorithm: configuration	1	1856 ✈̸	1869 ✈̸	1400 ✈̸
`flowGyroData`	threading	sensors: communication	2	4758 ✈̸	4130 ✈̸	3954 ✈̸
`motorRatioDef`	–	motors driver/low-level drivers	4	157 ✈̸	192 ✈̸	218 ✈
`simUpdate`	algorithms & logic	algorithm: wrong estimation	37	134 ✈	134 ✈	214 ✈
`byteSwap`	–	sensors: connectors/config.	3	2841 ✈̸	134 ✈	210345 ✈̸
`gyroAxesSwap`	–	sensors: connectors/config.	2	13464 ✈̸	134 ✈	272562 ✈̸
`timingKalman`	threading	algorithms: missed deadlines	2	172849 ✈̸	139 ✈	200 ✈
`flowDeckdtTiming`	threading	software: computer vision	7	131575 ✈̸	136 ✈	202 ✈
`slowTick`	configuration	platform: controller board	7	133 ✈	135 ✈	245 ✈̸

Steinbauer et al. [62] surveyed the participants to the RoboCup[30] competition about faults encountered during the robot development. The practitioners were asked about faults concerning: the robotic platform, the sensors, the control hardware (where “control” refers to the communication with a master device that monitors and provides commands), sensors, robot software (the control software), and algorithms. Among those components we exclude the control hardware since, as mentioned, “control” is used with a different meaning than in this work, and refers to the user interface. For each of the remaining we report the main sources of faults mentioned by developers:

- platform: batteries, motor drivers, and controller board,
- sensors: connectors, configuration, and communication,
- robot software: computer vision, inter-robot communication, and low-level device drivers,
- algorithms: configuration, wrong estimation, and missed deadlines.

Among those fault types we exclude “inter-robot communication” since we consider a single system.

We manually develop and inject faults on the base of the descriptions and examples of the categories mentioned above. We cover all of the categories listed by the two surveys that relate to control software. Table 2 reports the list of the developed faults: the second and third columns map them to the different categories of the surveys mentioned above [62, 71]. For each fault, we provide a patch file that injects it in the software [1].[31] After injecting a fault, we perform tests in the SIL, HIL and RIL setups with the same flight sequence from Figure 4. The drone software used is the same in each setup, ensuring consistent injection of the fault. By setting one compilation macro (respectively `SOFTWARE_IN_THE_LOOP`, `HARDWARE_IN_THE_LOOP`, and `PROCESS_IN_THE_LOOP`) the code is compiled for the desired setup.

5 EXPERIMENTAL RESULTS

In this section we describe the results of the tests performed with the injected faults. Most importantly, we identify for each fault the reasons why it appears or not in the different setups. In Section 6, we comment our tests and discuss the research questions in light of our results.

Table 2 reports the test results for each injected fault and each setup. We report whether the fault affects flight performance (✈̸) or not (✈) in the corresponding setup both qualitatively and quantitatively. On the qualitative side, we manually compare the traces of the flights with injected faults to the nominal behaviour observed in Figure 4. On the quantitative side, we compute the Integrated Squared Error (ISE), which is the sum over time of the squared difference between the desired position and the actual position (with the exception of the RIL where the ground truth is not available and we used the estimated position). The ISE is a standard metric for designing and evaluating control systems both in industrial contexts as well as in scientific literature (e.g., in the context of the PID controllers autotuning procedures) [7, 21]. While this metric does not directly relate to the fulfilment of a specific requirement, the violation of the most common requirements in control systems (e.g., settling time, static tracking, overshoot) will cause an increase in the ISE. For example, a higher settling time (i.e., time that the actual position takes to converge to its desired value) implies a larger error for longer time, hence an increase in the ISE. Therefore, we consider as impaired the flights that show an ISE higher that the one measured in the tests performed with the nominal software. For the SIL and HIL setups, we expect a deterministic result from the tests, hence, if a fault does not affect the drone flight, the ISE should have the same value. Accordingly, every increase in the ISE can be considered as a consequence of the injected fault. However, given the complexity of the simulator, we accept ISE changes in the order of units as, in practice, they do not correspond to visible differences in the flight performance. In the case of the RIL, we allow for some discrepancy given the variability involved in

[30] https://www.robocup.org/

[31] The repository contains information about the specific software version that we used, together with detailed instructions on how to retrieve the correct version and to inject the faults.

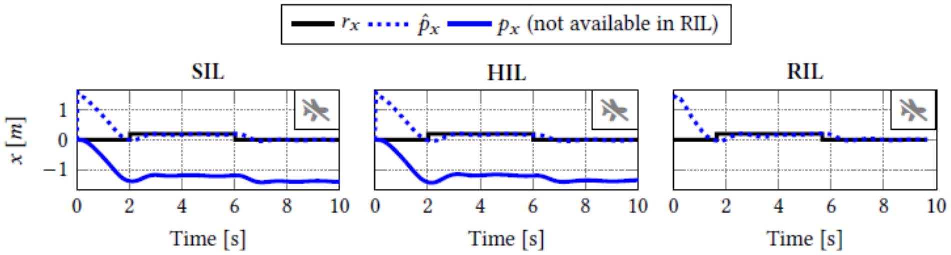


Fig. 5. SIL, HIL and RIL plots of x-axis position with the `initialPos` fault. This fault affects the initialization of the state estimator and appears equivalently in each setup. These tests show that faults in the functional aspects of the software appear equivalently across the setups.

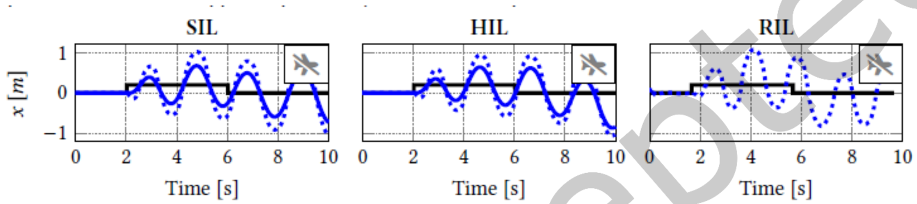


Fig. 6. SIL, HIL and RIL plots of x-axis position with the `flowGyroData` fault. This fault affects the fusion of the inertial and visual odometry data and appears equivalently in each setup, more specifically the optical flow data with the attitude rate. These tests show that faults in the functional properties of the software appear equivalently across the setups.

the tests execution. We observe in Figure 4 that the RIL tests show a variability in the drone position (observed as the thickness of the shaded area) of up to ten centimetres at a given point in time. For such reason, given that the tests are ten seconds long and that the ISE metric is cumulative over time (hence depends on the test duration), we accept an increase of up to 20 units for the RIL tests.

We report that the qualitative and quantitative evaluations give consistent results. The only exception are two HIL tests, where we observe an increase of some units in the ISE of the `timingKalman` and `flowDeckdtTiming` tests: respectively of 4 and 1 units. While technically being an increase in the ISE metric, those value variations do not correspond to any visible change in the plots, hence we considered those tests as not impaired. Complete flight data and pre-generated plots are available in the associated repository [1], respectively inside the `flightdata` and `pdf` subfolders for each setup. We also report here plots for the faults that did not cause an immediate crash and are different from the nominal flights (Figures 5, 6, 7, and 8). In these plots, we show only the position along the x-axis, as it suffices our discussion. As for Figure 4, we show the reference value r_x together with the ground truth p_x and its estimated value $\hat{p}_x$.

We now describe each fault and analyse the reasons why it appears or not in our different setups. This is necessary to assess if the differences in fault exposure that we observe are due to our specific setups implementation choices or are associated to intrinsic properties of the setups. Such analysis is therefore fundamental to discuss the generalisability of our findings to other CPS. For example, as mentioned in Section 4.2, the use of dedicated hardware in the HIL setups can enable better coverage of low-level drivers — as experienced and reported below in the fault `motorRatioDef`. Therefore, as we did not develop custom hardware for this experimental campaign, we have to assess if and how it would have changed the results.

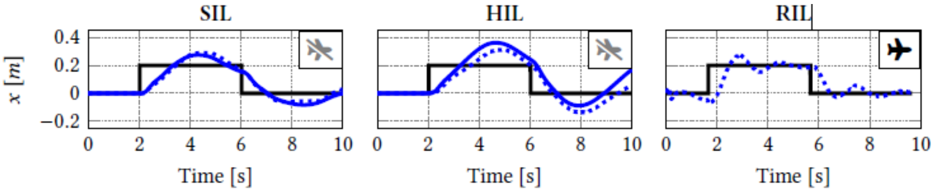


Fig. 7. SIL, HIL and RIL plots of x-axis position with the motorRatioDef fault. This fault is caused by different software components assuming different definitions of the same quantity: in this case the motor actuation value. As it affects the interaction with the low-level driver of the motors, it does not appear in all setups, in fact it does not affect the flight at the RIL level. This shows that abstract setups can cause false positives.

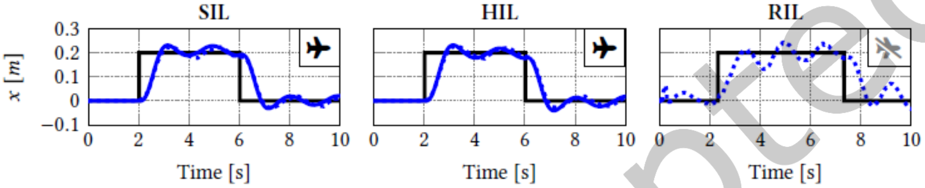


Fig. 8. SIL, HIL and RIL plots of x-axis position with the slowTick fault. This fault affects in the timing in the real-time operative system. As the timing of the code execution is abstracted in different ways it appears differently in the setups. Specifically it shows a limitation in capturing timing behaviour both in SIL and HIL.

The voltageCompCast and initialPos faults are taken from the Bitcraze repository[32]. The first consists of casting a float always smaller than 1 to an integer, which is then always rounded to zero. The variable contains the normalised motor commands: the control action is therefore always zero and the drone never takes off. The second fault concerns the wrong initialisation of the state estimator. In particular, the position estimate along the x axis is initialised to $1.5\,m$ instead of $0\,m$. Since no absolute position measurements is available, the state estimator cannot recover from this error. As shown in Figure 5, the controller reacts to the wrong estimation and brings the drone back to the presumed 0 position, which however is not the actual 0 position. This happens equivalently in each of our setups, hence showing that our testing abstractions do not alter the detection of these faults.

The flowGyroData fault alters how the estimator compensates for the angular rotation in the optical flow generated measurements. The code uses a local variable containing the latest gyro measurement. In the altered version, the code uses instead another queue containing the same information. However, the queue is also accessed in other parts of the code, making the data inconsistent at times. Figure 6 shows the results of the drone flight for each of our setups. The injected fault causes an error in the estimation of the speed and consequent large oscillations during the flight. This fault, like the two above, appears equivalently across the setups. It is interesting to note that this fault affects the functional properties of the control code since the equations of the state estimator are distorted. However, the behaviour of the drone is very similar among both the setups that include the simulated physics and the RIL. This means that, the abstraction introduced by the use of the physics

[32] initialPos: https://github.com/bitcraze/crazyflie-firmware/issues/760, and voltageCompCast: https://github.com/bitcraze/crazyflie-firmware/issues/766

model do not alter the impact of this fault on the drone behaviour. It can be noted however, that the oscillations resulting from the fault have a slightly different frequency, being faster in RIL with respect to SIL and HIL.

The motorRatioDef fault consists of two parts of the software assuming different definitions for the variable containing the command signal to the motors.[33] In the altered software, the variable is implemented as a float smaller than 1 (a percentage) but read as an integer containing the actual command to the motors. This variable (which was not introduced by us and belongs to the original software) is read in the SIL and HIL setups, to detect if the drone is in flight or not. This information is used by the state estimator to compensate ground contact forces when the drone is not flying. Figure 7 shows how this affects the flight for the position along the x direction in our setups. In the RIL setup, the motor commands are directly read from the motor's hardware, hence the flight is not disturbed. This discrepancy is caused therefore by the abstraction of the motor hardware in our SIL and HIL setups. It could be avoided in SIL with the implementation of a more detailed emulator of the motors hardware and in HIL with the use of custom hardware.

The simUpdate fault concerns the incorrect implementation of different controller equations in the state estimator. When updating a vector with a matrix multiplication, the old vector values are not stored in a temporary variable, and the new values of the already updated components are used in place of the previous ones, as per specification. In a correct implementation, the software needs to store and use the previous vector values and use those to perform the update. In each testing setup, the controller code is robust to this fault. The obtained plots (reported in the associated repository [1]) are not distinguishable from the plots form Figure 4. As in flowGyroData, this fault that distorts the implementation of the control algorithm equations and alters the functional properties of the control software.

The byteSwap and gyroAxesSwap faults are injected in the low-level software that handles the interaction with the IMU. The former swaps the least and most significant bytes in the accelerometer readings. The latter swaps the x and y axes in the gyroscope readings. Both faults disrupt the flight in SIL and RIL, causing immediate crash. Conversely, the HIL setup is not affected and the fault is not detected. In the HIL setup, in fact, the IMU readings are injected at a higher level than in SIL. In the SIL in fact, as mentioned in Subsection 4.2, since the IMU is fully emulated the low-level firmware is not altered — only the calibration is skipped, but not the real-time sensor reading. Differently, in the HIL, the firmware stack that interacts with the sensors is interrupted in order to feed the values from the simulator to the software, in place of the actual sensor readings. As a result, the section of code where the faults are injected is not executed with the HIL setup, and the fault does not affect the flight. This is caused by the abstraction in the HIL setup of the low-level software used for communicating with the IMU. The use of dedicated hardware could avoid this abstraction in our HIL setup.

Finally, the timingKalman, flowDeckdtTiming and slowTick faults concern the timing of the code execution and its real-time properties. In the nominal software, the periodic execution of the thread executing the state estimator is triggered by a semaphore released by the IMU interrupt that signals the availability of the data. In timingKalman, the thread is instead put to sleep for a time equal to its period. This sleep time is measured by the software tick of the RTOS. This is a poor real-time programming practice as it introduces jitter in the execution of the thread. The flowDeckdtTiming uses a different timer to measure the time interval over which the optical flow is measured. The optical flow provides a differential measure, hence it is highly dependant on its recording time, which needs to be measured. Rigorously speaking, this is not a fault, as long as the different timers are consistent, however, we use it to expose the different timing properties of the setups. Both these faults impair the flight in the SIL setup but not in HIL or RIL. In both cases, this is due to a distortion in the representation of time (and hence of timers) during the execution of the software in Renode (the hardware emulator used to implement the SIL). Better representation of time in the Renode emulator would reduce the impact of the abstraction of the execution

[33] This fault is inspired by the episode of the NASA Mars Climate Orbiter that crashed in 1999. In that case, one software component assumed that a variable containing a pressure value was defined in Imperial Units while another used the International System of Units.

timing of the software. Achieving this is however a challenging task, and high-fidelity emulation of software execution time is an open research problem. In slowTick, a hardware clock malfunction is simulated by setting the RTOS software tick to 800 Hz instead of 1000 Hz. In our tests this does not affect our implementations in SIL and HIL setups but, as shown in Figure 8, it does impair flight in RIL. In the SIL and HIL setups, the simulated physics is timed by the RTOS main clock and hence the flow of time is still consistent between the controller and the simulated physics despite the injected fault. In the RIL setup, the physics evolves with the actual time, and the execution of the controller is therefore disturbed. The abstraction of the synchronization of time evolution in the execution of the software and in the evolution of the physics is at the base of this discrepancy.

6 DISCUSSION AND LIMITATIONS

In this section, we use the test results to address the research questions presented in Section 1. We use the analysis of each fault to discuss how our answers generalise to other control systems. At the end of each answer we summarize the main take-away messages from our observations. We conclude this section discussing the limitations and threats to validity in our study.

RQ1: What are the differences between the testing abstractions with respect to their fault revealing capability? Our case study shows that *the testing abstractions achieve different coverage of software and timing properties, but are similarly effective when testing the software functional properties.*

In fact, the different testing setups equally expose the voltageCompCast, initialPos, flowGyroData, and simUpdate faults. These faults affect the functional properties of the software (i.e. the implementation of the control law) and do not alter the low-level interaction with the hardware nor the timing of the software execution. The main testing abstraction that directly relates to the functional properties is the model of the physics. We also note that such abstraction is always found in the abstract setups of control software. Our experiments show that this does not cause relevant differences in the exposure of functional faults with respect to the RIL setup. However, the use of different physical models might still impact the detection of functional software faults (this discussion is out of the scope of our work, some investigations in this direction can be found in the literature [61]).

Among the other faults, byteSwap, gyroAxesSwap, and motorRatioDef affect the interaction between software and hardware, while timingKalman, flowDeckdtTiming, and slowTick affect the software timing. Both byteSwap and gyroAxesSwap are hidden in the HIL abstraction setup. This is due to the abstraction in HIL of the low-level interaction with the IMU. Conversely, they are exposed in the SIL setup. Therefore, *for our system, the SIL setup shows better code coverage than HIL. This is originated by the necessity of manipulating, in the HIL setup, the low-level code so that it interacts with the simulated physics. Differently, in SIL, we fully emulate the interaction with the hardware, and hence expose also faults in the low-level code.* This suggests that the fault-revealing capabilities of HIL can be limited for low-level code — which has been reported to be prone to faults in robotic systems [62, 71]. However, the actual extension and relevance of such limitation, as well as its generalisation to other CPSs requires further investigation. As mentioned in Section 3.3, to improve the low-level code coverage of HIL, dedicated hardware could be produced. With dedicated HIL hardware, output commands could be read form the output ports, and artificial data can be fed using the dedicated input ports. Apparently, dedicated hardware prototype likely increases production costs. On the other hand, SIL testing also requires an effort to implement the sensors' emulation.

The fault motorRatioDef alters the flight in the SIL and HIL setups (Figure 7). This fault affects the low-level interaction with the motors, hence a component that is abstracted in both setups. In RIL, the variable containing the faulty value is only written to and never read. In SIL and HIL, the motor commands cannot be read directly from the hardware, and this variable is read instead. Despite not affecting the RIL flight, the variable does contain a faulty value. This can be interpreted in two equally valid ways: either that SIL and HIL are introducing a false positive (i.e., they fail a test that should pass), or that the difference between the testing abstractions is pointing to dead code. Which interpretation is valid depends on the specific application. Either ways this shows that *the abstraction of a component can not only potentially cause a false negative (i.e., hiding a fault), but also introduce a false positive (i.e., causing a failure when it shouldn't happen).*

The three faults `timingKalman`, `flowDeckdtTiming`, and `slowTick` affect the timing of the code execution, and expose the differences in the timing-related abstractions between the testing setups. Both `timingKalman` and `flowDeckdtTiming` disrupt the flight in SIL: this is inconsistent with the RIL (and HIL) tests where flight is successful. The SIL setup is therefore introducing false positives, showing limitations in the abstraction of the timing of the code execution. More specifically, the faults affect the synchronization of different parts of the code: respectively the estimator task and the time measurement of the optical flow readings with the rest of the control code. Due to the time distortion of SIL, this loss of synchronisation impairs the correct execution of the code and the flight performance is impaired. Since these changes do not affect HIL or RIL, we can conclude that the modelling of time in Renode is not accurate enough. In SIL, it could be possible to improve the timing aspect of hardware emulation, for example, by profiling the target architecture. However, this is not an easy task and it is rather an open research problem [64, 65].

The remaining timing-related fault, `slowTick`, shows a limitation that is common to both SIL and HIL. In both setups, the simulation of the physics is synchronised to the RTOS software tick. A distortion in the RTOS clock will therefore not impair the synchronisation between the control algorithm and the physical component of the system. In the RIL setup, it is possible to detect this fault, since there is no modelled physics, the physical part evolves according to the actual time, independently of the software execution. In this case, the abstraction of the real-world flow of time in the physical model causes false negatives in SIL and HIL. In HIL, it would be possible to develop a simulator of the physics that is executed in real time and doesn't need to be synchronized with the software execution: this would allow to expose the `slowTick` fault in the HIL setup. A similar solution could be implemented in SIL: however, the limitations mentioned above in the emulation of time aspects of hardware execution would still hold.

These tests show that *abstracted testing setups will always have inherent limitation in the modelling of time, and this can significantly affect the quality of the control software testing process.* HIL setups have an advantage with respect to SIL setups, since they do not require explicit modelling of the timing of software execution as they include the target hardware. RIL does not require any abstraction and can provide time consistency (between software and physics) by definition.

On the other side, our tests also show that there are several aspects that speak in favour of complementing RIL with SIL and HIL. In the abstracted setups the physical world is simulated, and accessing the ground truth is always possible (e.g., the drone position in our case) while external sensors would be required for the RIL setup. Further considerations are related to the practical execution of the tests. SIL and HIL tests are fully reproducible, reducing the occurrence of flaky tests [36]; moreover, they are more easily automated and performed remotely.

Our observations support the following considerations on the fault-revealing capabilities of the different testing setups:

- functional properties appear equivalently across the setups,
- depending on the setups implementation choices, SIL can potentially expose better low-level code coverage with respect to HIL,
- depending on the setups implementation choices, HIL can provide better representation of the code execution timing, with respect to SIL,
- abstractions in the testing setups can cause both false negative (hiding faults) and false positive (failing tests that should pass).

RQ2: When and why is it beneficial to have different testing setups? What are the principles to be followed when designing the testing setups? We have seen that the use of different testing setups improves the testing coverage. However, *the exposure of a higher faults number is originated by having different abstractions in the testing setups, that therefore rely on different assumptions.*

As discussed in Section 1, previous literature assumed that the setups are hierarchically ordered and that the faults that can be found at a level of abstraction are a superset of the faults that can be found in a less abstract setup. However, our tests disprove this statement and show that the faults detectable in a setup are neither a subset nor a superset of the ones found in another setup. Accordingly, the best practice is to *maximise the difference between the testing abstractions of the available setups, to enhance testing coverage and fault finding.* Said in other words, it is not best to have every setup as detailed as possible (i.e., with minimal number of abstractions), rather it is important that the abstractions overlap between the different setups is minimal.[34]

The `flowGyroData` fault is discovered in all different testing setups. This suggests that the fault is related to a behaviour that is not abstracted in any of them. Because the testing abstractions are not the same across the setups, we can narrow the scope of the search for the fault and exclude all components that are abstracted in each of the setups. In our case we can deduce that the fault is not (among others) in the low-level interaction with the sensors and actuators, nor in the timing aspects of the code.

Conversely, `byteSwap` and `gyroAxesSwap` are detected in SIL and not in HIL. This suggests that they are faults related to the low-level IMU firmware which is executed in SIL but not in HIL. Suppose that the HIL tests were performed with dedicated hardware (as mentioned in Section 3.3) to enable the coverage of the IMU low-level firmware and detect faults like `byteSwap` and `gyroAxesSwap`. In this case, it would be more difficult to root cause the failure and identify the fault.

When designing the different testing setups the objective shall be to maximise the differences in the testing abstractions across the different setups rather than focusing on making each setup detailed. The natural choice is to focus on the strengths of each setup pointed out in the answer to RQ1. This will improve the fault identification process.

RQ3: What are the domain-specific characteristics of system testing for closed loop control software? System testing is clearly an important step in the development of any software [23]. On top of the general considerations on system testing and the motivation of this paper of tight coupling, our case study highlights challenges that specifically belong to system-level testing of control software. In particular, we conclude that *control systems expose robustness to software faults*, and *couple functional and non-functional properties, especially with respect to timing.*

In our example, the `simUpdate` tests show that, despite the fault, the drone is able to fly without decreased performance. This is not surprising, as control systems possess a certain level of robustness to the distortions that appear in their software implementation [8]. Said robustness varies depending on the physical component under control and its characteristics, as well as the specific control algorithm used. While robustness is a desirable property in the final product, in the `simUpdate` case, the code fails to match its specifications (i.e., it does not implement the prescribed equations). Consequently, the control theoretical guarantees cease to apply for a general flight and may be lost in certain operating conditions (e.g. in presence of wind). This poses the challenge of developing *adequate coverage metrics and test cases that enable the detection of faults that are hidden by closing the control loop.*

[34] From a practical point of view, it shall also be considered that testing in more abstract setups is usually less expensive, since more components are simulated. This is an important consideration for practical applications, but the evaluation of setup development costs is out of the scope of this investigation.

The three faults `flowDeckdtTiming`, `timingKalman` and `slowTick` show the sensitivity of the software to its timely execution. This is a general property of control software and time modelling is extremely important in the setups for system testing of control systems. We formulate the research challenge of *synthesising requirements of time modelling for the system testing of control systems*. Failing to formulate and meet such requirements can hide faults (e.g. `slowTick`), or create false positives (e.g. `timingKalman` and `flowDeckdtTiming` in SIL).

Our experiments show the following domain-specific characteristics in control software:

- robustness to software faults, and
- coupling of functional properties with execution timing properties.

Limitations and threats to validity. Our analysis is based on a single case study. Hence, we report its properties that limit the generalisation of our conclusions (external validity). Then, we discuss the limitations of our research methodology (internal validity).

Other control systems may differ from the Crazyflie with respect to hardware platform, software architecture, dynamics of the physical component, development method, and system criticality. For example, the control software of an aeroplane runs on more powerful hardware, has redundant sensors, and is (most likely) distributed. This is different from the Crazyflie that does not have redundant sensors and has only one computational core dedicated to the critical computations.

The development of the software is also different in a plane, as regulations constrain the verification and validation process [48]. This is different from the open-source development of the Crazyflie software and hardware. Finally, physical components can have different dynamics. As a consequence, timing properties are more or less relevant, the robustness to software faults changes, and input and output signals are different in nature. From this point of view, small drones (like the Crazyflie) are a relevant case of study as their low weight (and consequently fast reaction) makes the timing properties of the software highly critical. Despite such differences, our observations focus on aspects that characterize the testing of any control systems, namely: modelling of time in the setups, synchronisation of the different components, emulation of hardware, testing of functional and non-functional properties, and abstraction of low-level software. Furthermore, there are significant commonalities across every control system: for example, every control system performs at constant time intervals the actions of sensing, computing and actuating, and every control algorithm developed with traditional control engineering is specified with differential or difference equations. Apparently, a complete generalisation of our observations still requires further experimental validation.

Concerning our research methodology, a possible limitation is that our discussion and observations are based on faults developed from descriptions of typical robotics faults from previous literature [62, 71]. This can affect the real-world validity of the injected faults. However, in this study we focus on the capabilities of different testing setups in finding different types of faults. Therefore, what really matters is the component that is affected and in which way it is affected. It is not a strict requirement that the fault per se is realistic. Rather what is important is that the implementation specifications of the component are not fulfilled.

We also performed manually the analysis of the causes why the different faults appear or not in the different setups. However, we developed each of the testing setups from scratch, which gives us high confidence that our understanding of their implementation and properties is adequate. Concerning the development of the setups, a limitation is that we developed our setups by the reverse-engineering of a pre-existing control system. In a production environment, the development of the testing setups is done in parallel with the development of the system. Implementing the testing setups together with the system can help to better tailor them to the specific system and may increase their specific coverage. However, our analysis is focused on the differences between the setups rather than on the development process, and we argue that the observed differences are related to fundamental properties of the setups rather than to how they are developed. Furthermore, we developed the setups in close contact with the engineers at Bitcraze and we discuss in the paper the potential alternatives in the design of our testing setups, together with their potential impacts on the results of the study.

Finally, we note that in our methodology, we execute only one test case (the step response) for each fault and setup. This is different from performing a complete testing campaign on the system and it could potentially alter which faults are detected. However, we note that step responses are common practice to evaluate control requirements: in fact, the most common control requirements (rise time, settling time, overshoot, static tracking. . .) are defined on the base of step responses [8]. Furthermore, we note that *all* of the differences in fault exposure observed across the testing setups are caused by intrinsic properties of the setups. For example, the `gyroAxesSwap` does not appear in HIL because the code affected by the fault is not executed by the setups. Therefore, we can state that the difference in the exposure of the fault is caused by a property of the setup rather than choice of the test. Thanks to this analysis, the choice of the test case does not consist of a limitation in our experimental campaign.

7 RELATED WORK

Recent research highlighted interesting research directions at the intersection of control and software engineering [11, 14]. In the control literature, Zimmer et al. [74] discuss a case study on the consequences of implementation choices for the control performance. A comprehensive book on model based testing for embedded systems is the one by Zander et al. [72]; another review can be found in the works by Garousi et al. [24], and by Chattopadhyay et al.[12].

Whilst testing control software is not a new field, the vast majority of previous work is focused either on the testing models (i.e., model based testing), or on applications, mainly in the fields of avionics [58, 70] and automotive [16, 17].

The concept of testing abstractions is discussed in the book on model based testing by Zander et al. [72], and the testing setups discussed in this paper appear in different work, with slightly varying but overall consistent definitions [17, 32, 72]. The MIL setup has been extensively leveraged in the literature of model based testing [15]. The research has been focused on: verification of requirements [54], generation of test traces [28], or the use of models for the automatic generation of test cases with search algorithms [38, 40, 41], classification trees [32], system-identification based refinements [44], or genetic algorithms [3]. In robotics, Silano et al. [59] showcased the usefulness of SIL for the design of quadcopter controllers. Keranen et al. [29] perform a validation of model based approaches in the context of HIL testing, and Hansen et al. [27] do the same in the context of SIL testing. Meedeniya et al. [43] propose an optimisation for reliable deployment of control software. Ore et al. [57] propose to use program analysis to enrich of physics simulation and better test control software.

The papers above focus on individual setups, and we found the conclusions drawn in the literature compatible with ours. Despite the common industrial use of different testing levels, to the best of the authors' knowledge, the only other work that compares different testing setups is a very recent paper by Stocco et al. [63]. This latter work empirically evaluates how the use of simulated physics impacts the exposure of failures compared to a real-world testing setup. Framed in the jargon of our work, the work by Stocco et al. focuses how the abstractions associated with the use of simulated physics instead of the real world impact the exposition of failures. Differently, here we discuss the different testing abstractions that characterise each testing setup. The two papers are therefore complementary, as our discusses the different abstractions of the various setups and the work by Stocco et al. focuses on the abstractions related to the physics simulation.

8 CONCLUSION

In this paper, we provided a comparison of the fault-finding characteristics of the model-in-the-loop, software-in-the-loop, hardware-in-the-loop and real-physics-in-the-loop testing setups for the system-level testing of control systems. We presented the case study of an open-source drone and developed testing support for all the mentioned testing setups. We provide a complete replication package that enables further research on the topic (generally limited by the high implementation cost of different setups).

In order to investigate the differences across the setups, we injected different types of faults in the drone software developed on the base of descriptions of common faults by practitioners. Contrary to previous literature, we demonstrated with our case study that a hierarchy among these setups and abstractions does not exist. In other words, it is not necessarily true that testing setups closer to the real implementation can expose more bugs than the setups that rely on more abstractions. We evidenced that SIL setups are superior with respect to HIL in terms of low-level code coverage. Conversely, HIL better cover the timing properties of the code. On the other side, our experiments did not show major differences in terms of exposure of functional faults. We also highlighted the relevant properties and principles that have to be discussed by practitioners in the design of the testing setups. We evidenced that maximizing variety in the testing abstractions of the different setups (instead of minimising the abstractions in each setup) will enhance the testing process in terms of system coverage and fault identification.

ACKNOWLEDGEMENTS

Part of this work was performed while Max Nyberg Carlsson was conducting an internship at Bitcraze. We thank Josefine Möllerström, for being part of the SIL setup development, and the engineers at Bitcraze, in particular Marcus Eliasson, for the support with the Crazyflie platform and the insights on the development of the testing setups. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project). This publication reflects only the authors' view and the European Commission is not responsible for any use that may be made of the information it contains. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. At the time of doing this work, the authors were members of ELLIIT Strategic Research Area at Lund University.

REFERENCES

[1] https://github.com/ManCla/testing-abstractions. 2022.

[2] Afsoon Afzal, Claire Le Goues, Michael Hilton, and C. Timperley. 2020. A Study on Challenges of Testing Robotic Systems. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2020), 96–107.

[3] Aldeida Aleti and Lars Grunske. 2015. Test data generation with a Kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software* 103 (2015), 343–352. https://doi.org/10.1016/j.jss.2014.11.035

[4] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. 2019. Some Challenges for Software Testing Research (Invited Talk Paper). In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3293882.3338991

[5] Brandon Araki, John Strang, Sarah Pohorecky, Celine Qiu, Tobias Naegeli, and Daniela Rus. 2017. Multi-robot path planning for a swarm of robots that can both fly and drive. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*. IEEE, 5575–5582. https://doi.org/10.1109/ICRA.2017.7989657

[6] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. 2018. A Runtime Verification Tool for Detecting Concurrency Bugs in FreeRTOS Embedded Software. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*. 172–179. https://doi.org/10.1109/ISPDC2018.2018.00032

[7] Karl Johan Åström and Tore Hägglund. 2006. *Advanced PID Control*. ISA - The Instrumentation, Systems and Automation Society.

[8] Karl Johan Åstrom and Richard M. Murray. 2008. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA.

[9] Karl J Åström and Björn Wittenmark. 2013. *Computer-controlled systems: theory and design*. Courier Corporation.

[10] Johannes Bach, Jacob Langner, Stefan Otten, Eric Sax, and Marc Holzäpfel. 2017. Test scenario selection for system-level verification and validation of geolocation-dependent automotive control systems. In *2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. 203–210. https://doi.org/10.1109/ICE.2017.8279890

[11] Balaji Balasubramaniam, Hamid Bagheri, Sebastian Elbaum, and Justin Bradley. 2020. Investigating Controller Evolution and Divergence through Mining and Mutation*. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*. 151–161. https://doi.org/10.1109/ICCPS48487.2020.00022

[12] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2016. Chapter Three - On Testing Embedded Software. Advances in Computers, Vol. 101. Elsevier, 121–153. https://doi.org/10.1016/bs.adcom.2015.11.005

[13] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. 2021. A Survey of Field-Based Testing Techniques. *ACM Comput. Surv.* 54, 5, Article 92 (May 2021), 39 pages. https://doi.org/10.1145/3447240

[14] Justin M. Bradley and Hamid Bagheri. [n. d.]. *Control Software: Research Directions in the Intersection of Control Theory and Software Engineering*. https://doi.org/10.2514/6.2020-2102 arXiv:https://arc.aiaa.org/doi/pdf/10.2514/6.2020-2102

[15] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. 2016. Testing the Untestable: Model Testing of Complex Software-Intensive Systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 789–792. https://doi.org/10.1145/2889160.2889212

[16] Eckard Bringmann and Andreas Krämer. 2006. Systematic Testing of the Continuous Behavior of Automotive Systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems (SEAS '06)*. Association for Computing Machinery, New York, NY, USA, 13–20. https://doi.org/10.1145/1138474.1138479

[17] Eckard Bringmann and Andreas Krämer. 2008. Model-Based Testing of Automotive Systems. In *2008 1st International Conference on Software Testing, Verification, and Validation*. 485–493. https://doi.org/10.1109/ICST.2008.45

[18] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. 2007. Engineering Automotive Software. *Proc. IEEE* 95, 2 (Feb 2007), 356–373. https://doi.org/10.1109/JPROC.2006.888386

[19] Barbara Barros Carlos, Tommaso Sartor, Andrea Zanelli, Gianluca Frison, Wolfram Burgard, Moritz Diehl, and Giuseppe Oriolo. 2020. An Efficient Real-Time NMPC for Quadrotor Position Control under Communication Time-Delay. In *16th International Conference on Control, Automation, Robotics and Vision, ICARCV 2020, Shenzhen, China, December 13-15, 2020*. IEEE, 982–989. https://doi.org/10.1109/ICARCV50220.2020.9305513

[20] Lane Desborough and Randy Miller. 2002. Increasing Customer Value of Industrial Control Performance Monitoring — Honeywell ' s Experience.

[21] Richard Dorf and Robert Bishop. 2017. *Modern Control Systems, 13th Edition*.

[22] Julian Förster.. 2015-08. System Identification of the Crazyflie 2.0 Nano Quadrocopter.

[23] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. 2020. Robotics Software Engineering: A Perspective from the Service Robotics Domain. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3368089.3409743

[24] Vahid Garousi, Michael Felderer, Çağrı Murat Karapıçak, and Uğur Yılmaz. 2018. Testing embedded software: A survey of the literature. *Information and Software Technology* 104 (2018), 14–45. https://doi.org/10.1016/j.infsof.2018.06.016

[25] Iris Gräßler, Eric Bodden, Jens Pottebaum, Johannes Geismann, and Daniel Roesmann. 2020. Security-Oriented Fault-Tolerance in Systems Engineering: A Conceptual Threat Modelling Approach for Cyber-Physical Production Systems. In *Advanced, Contemporary Control*, Andrzej Bartoszewicz, Jacek Kabziński, and Janusz Kacprzyk (Eds.). Springer International Publishing, Cham, 1458–1469.

[26] Marcus Greiff.. 2017. Modelling and Control of the Crazyflie Quadrotor for Aggressive and Autonomous Flight by Optical Flow Driven State Estimation. Student Paper.

[27] N. Hansen, Norbert Wiechowski, Alexander Kugler, S. Kowalewski, Thomas Rambow, and R. Busch. 2017. Model-in-the-Loop and Software-in-the-Loop Testing of Closed-Loop Automotive Software with Arttest. In *GI-Jahrestagung*.

[28] Joachim Hänsel, Daniela Rose, Paula Herber, and Sabine Glesner. 2011. An Evolutionary Algorithm for the Generation of Timed Test Traces for Embedded Real-Time Systems. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 170–179. https://doi.org/10.1109/ICST.2011.37

[29] Janne Keränen and Tomi Räty. 2013. Validation of Model-Based Testing in Hardware in the Loop Platform. In *2013 10th International Conference on Information Technology: New Generations*. 331–336. https://doi.org/10.1109/ITNG.2013.53

[30] Pierre Laclau, Vladislav Tempez, Franck Ruffier, Enrico Natalizio, and Jean-Baptiste Mouret. 2021. Signal-Based Self-Organization of a Chain of UAVs for Subterranean Exploration. *Frontiers Robotics AI* 8 (2021), 614206. https://doi.org/10.3389/frobt.2021.614206

[31] Frank Lamb. 2013. *Industrial Automation: Hands On*. McGraw-Hill Professional, US. https://doi.org/10.1036/9780071816472

[32] Klaus Lamberg, Michael Beine, Mario Eschmann, Rainer Otterbach, Mirko Conrad, and Ines Fey. 2004. Model-based Testing of Embedded Automotive Software Using Mtest. In *SAE 2004 World Congress and Exhibition*. SAE International. https://doi.org/10.4271/2004-01-1593

[33] Edward Ashford Lee and Sanjit Arunkumar Seshia. 2016. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* (2nd ed.). The MIT Press.

[34] William S. Levine. 2009. *The Control Systems Handbook* (2nd ed.). CRC Press, Inc., USA.

[35] D. Lohar, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and M. Christakis. 2021. A Two-Phase Approach for Conditional Floating-Point Verification. *Tools and Algorithms for the Construction and Analysis of Systems* 12652 (2021), 43 – 63.

[36] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[37] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. 2019. Dragonfly: a Tool for Simulating Self-Adaptive Drone Behaviours. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 107–113. https://doi.org/10.1109/SEAMS.2019.00022

[38] Bogdan Marculescu, Robert Feldt, Richard Torkar, and Simon Poulding. 2015. An initial industrial evaluation of interactive search-based testing for embedded software. *Applied Soft Computing* 29 (April 2015), 26–39. Issue 0. https://doi.org/10.1016/j.asoc.2014.12.025

[39] Abel Marrero Perez and Stefan Kaiser. 2009. Integrating Test Levels for Embedded Systems. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*. 184–193. https://doi.org/10.1109/TAICPART.2009.22

[40] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Brcukmann. 2014. MiL Testing of Highly Configurable Continuous Controllers: Scalable Search Using Surrogate Models. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/2642937.2642978

[41] Reza Matinnejad, Shiva Nejati, and Lionel C. Briand. 2017. Automated Testing of Hybrid Simulink/Stateflow Controllers: Industrial Case Studies. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 938–943. https://doi.org/10.1145/3106237.3117770

[42] James Clerk Maxwell. 2011. *On Governors*. Cambridge Library Collection - Physical Sciences, Vol. 2. Cambridge University Press, 105–120. https://doi.org/10.1017/CBO9780511710377.009

[43] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. 2011. Reliability-driven deployment optimization for embedded systems. *Journal of Systems and Software* 84, 5 (2011), 835–846. https://doi.org/10.1016/j.jss.2011.01.004

[44] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. 2019. Approximation-Refinement Testing of Compute-Intensive Cyber-Physical Models: An Approach Based on System Identification. *CoRR* abs/1910.02837 (2019). arXiv:1910.02837 http://arxiv.org/abs/1910.02837

[45] C. Menghi, P. Spoletini, M. Chechik, and C. Ghezzi. 2019. A verification-driven framework for iterative design of controllers. *Formal Aspects of Computing* (2019), 1–44.

[46] Derek Mitchell, Ellen A. Cappo, and Nathan Michael. 2016. Persistent robot formation flight via online substitution. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*. IEEE, 4810–4815. https://doi.org/10.1109/IROS.2016.7759706

[47] Daniel K. Molzahn, Florian Dörfler, Henrik Sandberg, Steven H. Low, Sambuddha Chakrabarti, Ross Baldick, and Javad Lavaei. 2017. A Survey of Distributed Optimization and Control Algorithms for Electric Power Systems. *IEEE Transactions on Smart Grid* 8, 6 (2017), 2941–2962. https://doi.org/10.1109/TSG.2017.2720471

[48] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. 2013. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software* 30, 3 (2013), 50–57. https://doi.org/10.1109/MS.2013.43

[49] Mark W Mueller, Michael Hamer, and Raffaello D'Andrea. 2015. Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 1730–1736. https://doi.org/10.1109/ICRA.2015.7139421

[50] Mark W Mueller, Markus Hehn, and Raffaello D'Andrea. 2016. Covariance Correction Step for Kalman Filtering with an Attitude. *Journal of Guidance, Control, and Dynamics* (2016), 1–7.

[51] Mark W. Müller and Raffaello D'Andrea. 2014. Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*. IEEE, 45–52. https://doi.org/10.1109/ICRA.2014.6906588

[52] Anitha Murugesan, Sanjai Rayadurgam, Michael W. Whalen, and Mats P. E. Heimdahl. 2015. Design Considerations for Modeling Modes in Cyber–Physical Systems. *IEEE Design Test* 32, 5 (2015), 66–73. https://doi.org/10.1109/MDAT.2015.2462112

[53] Hausi A. Müller. 2017. The Rise of Intelligent Cyber-Physical Systems. *Computer* 50, 12 (2017), 7–9. https://doi.org/10.1109/MC.2017.4451221

[54] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. 2019. Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1015–1025. https://doi.org/10.1145/3338906.3340444

[55] Luan V. Nguyen, Khaza Anuarul Hoque, Stanley Bak, Steven Drager, and Taylor T. Johnson. 2018. Cyber-Physical Specification Mismatches. *ACM Trans. Cyber-Phys. Syst.* 2, 4, Article 23 (July 2018), 26 pages. https://doi.org/10.1145/3170500

[56] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D. Ames, Jessy W. Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. 2016. Correct-by-Construction Adaptive Cruise Control: Two Approaches. *IEEE Transactions on Control Systems Technology* 24,

4 (2016), 1294–1307. https://doi.org/10.1109/TCST.2015.2501351

[57] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2018. Towards Code-Aware Robotic Simulation: Vision Paper. In *Proceedings of the 1st International Workshop on Robotics Software Engineering (RoSE '18)*. Association for Computing Machinery, New York, NY, USA, 40–43. https://doi.org/10.1145/3196558.3196566

[58] Jan Peleska. 2002. Hardware/Software Integration Testing for the new Airbus Aircraft Families. *http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html* (01 2002). https://doi.org/10.1007/978-0-387-35497-2_24

[59] Giuseppe Silano, Emanuele Aucone, and Luigi Iannelli. 2018. CrazyS: A Software-In-The-Loop Platform for the Crazyflie 2.0 Nano-Quadcopter. In *2018 26th Mediterranean Conference on Control and Automation (MED)*. 1–6. https://doi.org/10.1109/MED.2018.8442759

[60] Nathanial T. Smith, James T. Heineck, and Edward T. Schairer. [n. d.]. *Optical Flow for Flight and Wind Tunnel Background Oriented Schlieren Imaging*. https://doi.org/10.2514/6.2017-0472 arXiv:https://arc.aiaa.org/doi/pdf/10.2514/6.2017-0472

[61] Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, and Félix Ingrand. 2017. Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 150–159. https://doi.org/10.1109/QRS.2017.25

[62] Gerald Steinbauer. 2013. A Survey about Faults of Robots Used in RoboCup. In *RoboCup 2012: Robot Soccer World Cup XVI*, Xiaoping Chen, Peter Stone, Luis Enrique Sucar, and Tijn van der Zant (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–355.

[63] Andrea Stocco, Brian Pulfer, and Paolo Tonella. 2023. Mind the Gap! A Study on the Transferability of Virtual Versus Physical-World Testing of Autonomous Driving Systems. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1928–1940. https://doi.org/10.1109/TSE.2022.3202311

[64] James R. Taylor, Evan M. Drumwright, and Gabriel Parmer. 2014. Making Time Make Sense in Robotic Simulation. In *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots - Volume 8810 (SIMPAR 2014)*. Springer-Verlag, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/978-3-319-11900-7_1

[65] James R. Taylor, Evan M. Drumwright, and Gabriel Parmer. 2014. Temporally Consistent Simulation of Robots and Their Controllers *(International Design Engineering Technical Conferences and Computers and Information in Engineering Conference)*, Vol. Volume 6: 10th International Conference on Multibody Systems, Nonlinear Dynamics, and Control. https://doi.org/10.1115/DETC2014-35609

[66] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues. 2018. Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 331–342. https://doi.org/10.1109/ICST.2018.00040

[67] R.M. van der Knijff. 2014. Control systems/SCADA forensics, what's the difference? *Digital Investigation* 11, 3 (2014), 160–174. https://doi.org/10.1016/j.diin.2014.06.007 Special Issue: Embedded Forensics.

[68] Gang Wang, Weixin Yang, Na Zhao, Yunfeng Ji, Yantao Shen, Hao Xu, and Peng Li. 2020. Distributed Consensus Control of Multiple UAVs in a Constrained Environment. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*. IEEE, 3234–3240. https://doi.org/10.1109/ICRA40945.2020.9196926

[69] Michael W. Whalen, Anitha Murugesan, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2014. Structuring Simulink Models for Verification and Reuse. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering (MiSE 2014)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/2593770.2593776

[70] A.L. White. 2001. Comments on modified condition/decision coverage for software testing [of flight control software]. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, Vol. 6. 2821–2827 vol.6. https://doi.org/10.1109/AERO.2001.931302

[71] Johannes Wienke, Sebastian Meyer zu Borgsen, and Sebastian Wrede. 2016. A Data Set for Fault Detection Research on Component-Based Robotic Systems. In *Towards Autonomous Robotic Systems*, Lyuba Alboul, Dana Damian, and Jonathan M Aitken (Eds.). Springer International Publishing, Cham, 339–350.

[72] Justyna Zander, Ina Schieferdecker, and Pieter Mosterman. 2011. *Model-Based Testing for Embedded Systems*.

[73] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. 2017. Perceptions on the State of the Art in Verification and Validation in Cyber-Physical Systems. *IEEE Systems Journal* 11, 4 (2017), 2614–2627. https://doi.org/10.1109/JSYST.2015.2496293

[74] Michael Zimmer, J. Hedrick, and Edward A. Lee. 2015. Ramifications of software implementation and deployment: A case study on yaw moment controller design. *2015 American Control Conference (ACC)* (2015), 2014–2019.