Testing Abstractions for Cyber-Physical Control Systems - RCR Report

CLAUDIO MANDRIOLI, University of Luxembourg, Luxembourg
MAX NYBERG CARLSSON, Lund Univeriesty, Sweden
MARTINA MAGGIO, Saarland Univeriesty, Germany and Lund University, Sweden

This is the Replicated Computational Results (RCR) Report for the article "*Testing Abstractions for Cyber-Physical Control Systems.*" The article empirically studies how substituting different components in Cyber-Physical Systems (CPSs) testing with simulators impacts the fault-exposition. This RCR report describes the artefacts used in the paper, how to use the testing setups used in the article, and how to reproduce the empirical results of the article.

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

ACM Reference Format:
Claudio Mandrioli, Max Nyberg Carlsson, and Martina Maggio. 2025. Testing Abstractions for Cyber-Physical Control Systems - RCR Report. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 7 pages. https://doi.org/10.1145/3736577

1 Overview

1.1 Article Motivation

Cyber-Physical Systems (CPS) are characterised by the closed-loop interaction of software, hardware, and physical components. This interaction limits the ability to test CPS components in isolation. To overcome this limitation, practitioners abstract different CPS components and substitute them with their simulated counterparts. For example, the physical component can be substituted with simulation models of the physics, or the hardware can be substituted with an emulator. Abstracting different components corresponds to implementing different testing setups. Such setups are usually called X-in-the-loop, where X (e.g., software or hardware) qualifies the components that are included and the ones that are abstracted. Previous literature on the topic studied testing approaches to individual testing setups, but there is limited literature that compares the different setups using benchmarks or case studies.

1.2 Article Contribution

Our article fills this research gap providing a general discussion of the testing abstractions in control systems' testing, and empirically comparing the most common testing setups in terms of their ability to detect different types of software faults [1]. The empirical comparison addresses the following research questions (RQs):

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

ACM 1557-7392/2025/1-ART1
https://doi.org/10.1145/3736577

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?

To answer our RQs, we use a single case study, the Crazyflie drone, whose software and hardware are both open-source.[1] We equipped the drone with the Flow deck[2] that enhances the sensing capability of the drone by using a camera and a laser sensor. This allows the drone to fly autonomously without other external sensors. We developed three testing setups (in collaboration with Bitcraze, the company developing and maintaining the drone itself):

- Model-In-the-Loop (MIL), where the software is substituted with equation-based models of the control algorithms and interacts with an simulator of the physics.
- Software-In-the-Loop (SIL), where the software is executed on a hardware emulator and interacts with a simulator of the physics.
- Hardware-In-the-Loop (HIL), where the software is executed on the target hardware but interacts with a simulator of the physics running on a general-purpose machine.
- Real-world-In-the-Loop (RIL), where the drone actually flies.

We then test the SIL, HIL, and RIL setups after injecting ten different faults in the drone's firmware and study which setups expose the different faults. We excluded the MIL as it does not include the software, and only four faults could be replicated in this setup.

Answer to RQ1. Our experimental results show that the functional properties are equivalently tested across the setups. However, depending on implementation choices of the setup, we showed that (i) SIL can potentially provide better low-level code coverage with respect to HIL, and (ii) HIL can provide better representation of code execution timing. Furthermore, we observed that testing setups can cause both false negatives and false positives. Notably, this complementary nature of the setups is opposed to the hierarchical view often assumed or implied by previous literature.

Answer to RQ2. Based on our results, we recommend to design the testing setups to maximise the differences in the testing abstractions rather than to implement detailed setups. 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.

Answer to RQ3. Our experiments show that control software shows two domain-specific characteristics: (i) robustness to software faults, and (ii) coupling of functional properties with execution timing properties.

2 Artefact

We provide all the material necessary to implement the testing setups and to reproduce the paper results in a repository. The repository is available in the long-term archive Zenodo (https://doi.org/10.5281/zenodo.12820405). It is released with the permissive GNU GPL-3.0 license under the CC-BY (🅭🅯) copyright. The code can also be retrieved from GitHub.[3]

The repository is structured as follows:

- The *bugs* directory contains the patch files to inject the bugs in the drone's firmware.

[1]https://www.bitcraze.io/products/crazyflie-2-1/
[2]https://www.bitcraze.io/products/flow-deck-v2/
[3]https://github.com/ManCla/testing-abstractions/

Table 1. Summary of Hardware and Software dependencies. The ✚ symbol indicates that an item is needed for a given setup.

	Item	SIL	HIL	RIL	Retrieve at
Hardware	Bitcraze Crazyflie drone 2.1		✚	✚	Bitcraze
	Bitcraze Flow deck V2		✚	✚	Bitcraze
	Bitcraze debug adapter		✚		Bitcraze
	ST-LINK/V2 in-circuit debugger/programmer		✚		STM
	Bitcraze Micro SD card deck			✚	Bitcraze
Software	Python3.9.9	✚	✚	✚	Releases
	Crazyflie firmware	✚	✚	✚	Repository
	OpenOCD 0.11.0	✚	✚		Website
	Toolchain to build the firmware	✚	✚	✚	Bitcraze
	Bitcraze custom Renode repository	✚			Repository
	Bitcraze custom Renode Infrastructure repository	✚			Repository

- The *firmware* directory contains the firmware patch file for the Crazyflie firmware to implement the testing setups. This is also the directory where some of the compiled firmware files have to be placed when performing SIL and HIL testing.
- The *testing-frameworks* directory contains the main scripts to run the different testing setups and to plot the tests results. It also contains different sub-directories dedicated to each setup. Each sub-directory contains (i) the code implementing the functionalities specific to each setup, (ii) the pre-computed test results in the *flightdata* directories, and (iii) the pre-generated pdfs of the plots in the *pdf* directories.

3 Prerequisites and Requirements

The experiments are based on the Crazyflie drone. The MIL setup is built with only Python scripts. The SIL, HIL and RIL setups need building the firmware, and HIL and RIL setups also need flashing the drone: these actions require installing the toolchain outlined in the drone documentation.[4] The SIL setup runs on a general-purpose machine: we use Renode[5] to emulate the Crazyflie microprocessor and execute it in closed loop with the Python model of the physics. The HIL and RIL require the drone itself as well as other related pieces of hardware. The HIL setup uses, in terms of hardware, the ST-LINK debugger and the Bitcraze debug adapter to connect a general-purpose machine with the micro-controller of the drone. In terms of software, it uses OpenOCD (version 0.11.0). In the HIL setup, the drone then executes its firmware normally and interacts, through the debugger, with the Python model of the physics running on the connected machine. In the RIL setup, the drone is used as intended. In RIL tests, the drone is equipped with the Flowdeck (as mentioned in Section 1.2,); furthermore, to log the flight data, we also equip it with a Micro SD card. The complete list of required hardware and software is reported in Table 1. This table also reports which dependencies are specific to each testing setup (the SIL, HIL, and RIL columns), as well as the links where the hardware can be purchased and the software retrieved.

The code was developed and tested on MacOS 11.1 and Linux Fedora. All the main scripts are written in Python (we used Python 3.9.9).

[4]https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/building-and-flashing/build/
[5]https://renode.io

4 Setup Instructions

The setup of the experiments includes: cloning the artefact repository, retrieving the required version of the drone firmware, and compiling and setting up the custom version of Renode.

Cloning the Artefact

The artefact can be cloned from the GitHub repository with the following terminal command:

```
git clone git@github.com:ManCla/testing-abstractions.git
```

Alternatively, it can be downloaded from Zenodo.

Retrieving the Drone Firmware

The drone firmware can be cloned from the Bitcraze repository (note the `--recursive` option to clone also the submodules).

```
git clone --recursive git@github.com:bitcraze/crazyflie-firmware.git
```

To obtain the same version of the code that we used, check out to the commit 23e9b80 with the following commands:

```
cd crazyflie-firmware
git checkout 23e9b80
```

Finally, apply the patch provided in the artefact. Copy the patch file from the artefact repository to the firmware repository and then apply it with the following commands:

```
cp /PATH/testing-abstractions/firmware/firmware.patch \
/PATH/crazyflie-firmware/firmware.patch
git apply firmware.patch
```

Custom Renode Built and Setup

The SIL setup uses the custom Renode version maintained by Bitcraze in combination with the custom Renode Infrastructure (which is a submodule of Renode itself). First, clone Renode from the Bitcraze repository, check out to the *crazyflie* branch, and initialise the submodules

```
git clone git@github.com:bitcraze/renode.git
cd renode
git checkout crazyflie
git submodule init
git submodule update
```

The rest of the installation follows the standard Renode installation.

- Install dependencies according to Building Renode from source.
- Build Renode according to the Building Renode instructions.

Finally, we need to change the file that describes the Crazyflie hardware platform (as the default one does not include the Flowdeck) and tell Renode where to find the binaries to execute. In the Renode repository, change the file *renode/platforms/boards/cf2.repl* with the one provided in our artefact *testing-abstractions/testing-frameworks/sitl/cf2.repl*:

```
rm PATH/renode/platforms/boards/cf2.repl
mv PATH/testing-abstractions/testing-frameworks/sitl/cf2.repl \
PATH/renode/platforms/boards/cf2.repl
```

In the the Renode repository, in the file *renode/scripts/single-node/crazyflie.resc* change the path to the *.elf* compiled binary from

```
sysbus LoadELF @cf2.elf
```

to

```
sysbus LoadELF @/PATH/testing-abstractions/firmware/cf2.elf
```

5 Steps to Reproduce

We ran eleven tests for each of the three setups, for a total of thirty-three tests.[6] The eleven tests include: the test with the unaltered firmware (called "nominal tests") and one test for each of the ten faults. Reproducing the experiments includes

- injecting the fault (unless it is a nominal test),
- compiling the firmware for the target setup,
- running the target setup, and
- plotting the results.

5.1 Fault Injection

The faults are injected in the Crazyflie firmware using the patch files in the artefact's *bugs* directory. For example, to inject the `byteSwap` fault, copy the patch file to the firmware directory with

```
cp /PATH/testing-abstractions/bugs/byteSwap.patch \
/PATH/crazyflie-firmware/byteSwap.patch
```

Then, apply the patch to the code with

```
cd /PATH/crazyflie-firmware
git apply byteSwap.patch
```

5.2 Firmware Compilation

To compile the code, go to the directory of the test flight application

```
cd /PATH/crazyflie-firmware/examples/demos/app_steps/
```

and compile the code using the make target of the desired setup (`sitl` for SIL, `hitl` for HIL, and `pitl` for RIL). For example, for SIL use

```
make sitl
```

The firmware compilation generates different binaries in the directory *crazyflie-firmware/examples /demos/app_steps/*.

5.3 Running The Setups

Software-In-The-Loop. Among the binary files generated with the firmware compilation, move *cf2.elf* and *cf2.map* to the *testing-abstractions/firmware/* directory

```
mv /PATH/crazyflie-firmware/examples/demos/app_steps/cf2.elf \
/PATH/testing-abstractions/firmware/cf2.elf
mv /PATH/crazyflie-firmware/examples/demos/app_steps/cf2.map \
/PATH/testing-abstractions/firmware/cf2.map
```

To run the SIL test run, in two separate terminal windows, Renode and the main Python script executing the tests, respectively. From one terminal window, navigate to the Renode directory and start it:[7]

```
cd PATH/Renode
output/bin/Release/Renode.exe --disable-xwt --port 4444
```

[6]This number excludes the 30 repeated flights of the RIL setup without injected faults. We used those tests only to assess the repeatability of the RIL test flights.
[7]If the port 4444 is busy, select another one.

From another terminal window, navigate to the *testing-frameworks* directory and run the Python script[8]

```
cd PATH/testing-abstractions/testing-frameworks
python sitl_main.py <port>
```

Printouts of the test progress should now be appearing (note that each of these tests can take up to 12 hours to run). Once the test is finished the flight data are stored in a pickled file in the directory *testing-abstractions/testing-frameworks/sitl/flightdata* with a name corresponding to the date and time of the test end.

Hardware-In-The-Loop. Copy the *cf2.map* file to the *testing-abstractions/firmware* directory:

```
mv /PATH/crazyflie-firmware/examples/demos/app_steps/cf2.map \
/PATH/testing-abstractions/firmware/cf2.map
```

Then, prepare the hardware. First, mount the Flowdeck on the drone. Connect the drone to the computer with the debug adapter and the ST-Link on-chip debugger. We can now use the wired connection to flash the firmware on the drone. From the directory with the compiled binaries use the dedicated `flash` make target:

```
cd PATH/crazyflie-firmware/examples/demos/app_steps/
make flash
```

To run the HIL tests, we run OpenOCD to connect with the hardware, and the main Python script in two separate windows. From one window, navigate to the *crazyflie-firmware* directory and start OpenOCD with the dedicated make target

```
cd PATH/crazyflie-firmware
make openocd
```

After OpenOCD is connected to the hardware, from another terminal window in the same directory run the main script

```
cd PATH/testing-abstractions/testing-frameworks
python hitl_main.py
```

The test should now start and the test time progress should be displayed. These tests should each take less than a couple of hours to execute. Once the test is finished, the flight data are stored in a pickled file in the directory *testing-abstractions/testing-frameworks/hitl/flightdata* with a name corresponding to the date and time of the test end.

Real-World-In-The-Loop. Prepare the Micro SD card by copying the file *testing-levels/testing-frameworks/pitl/config.txt* in its root directory. Prepare the drone by mounting both the Flow deck and the Micro SD deck (remember to insert the card in it). Flash the firmware with the same procedure as the HIL setup: connect the drone to the computer with the debug adapter and ST-link and running `make flash` from the *crazyflie-firmware/examples/demos/app_steps/* directory. Note that after receiving the firmware, the drone will restart and start the test sequence after few seconds. While flashing the firmware, we recommend putting it upside down to preventing it from taking off. Then, turn it off and restart it from a safe location. After the flight, the Micro SD card contains the logged flight data. Use the *pitl_main.py* script to translate the data in the same format as the other test data. Provide the path to the log file as command line argument

```
cd PATH/testing-abstractions/testing-frameworks
python pitl_main.py PATH/LOG-FILE
```

[8]The optional argument `<port>` can be used if a different port was selected when starting Renode.

The processed flight data are stored in a pickled file in the directory *testing-abstractions/testing-frameworks/pitl/flightdata* with a name corresponding to the date and time at which the script was run.

5.4 Plotting the Flight Data

The results of any test can be plotted with the *plot_main.py* Python script

```
python plot_main.py show PATH/TO/FLIGHT-DATA
```

Reproducing Article Figures. The nominal plots (the ones in Figure 4 of the article) can be reproduced with the provided flight data executing the following command in a terminal window from the *testing-frameworks* directory (substitute `mitl` with `sitl` for the SIL test, with `hitl` for the HIL test, and `pitl` for the RIL test)

```
python plot_main.py show mitl/flightdata/nominal
```

Figures 5 to 8 of the article can be reproduced, respectively, with

```
python plot_main.py show */flightdata/initialPos
python plot_main.py show */flightdata/flowGyroData
python plot_main.py show */flightdata/motorRatioDef
python plot_main.py show */flightdata/slowTick
```

As above, substitute the asterisk with the directory of the setup from which you want to plot the results (`sitl` for the SIL test, `hitl` for the HIL test, and `pitl` for the RIL test).

Finally, the Integrated-Squared-Error (ISE) values used in Table 1 of the article can be computed with the *compute_error.py* script

```
python compute_error.py
```

Acknowledgments

This project has received funding from SES; the Luxembourg National Research Fund under the Industrial Partnership Block Grant (IPBG), ref. IPBG19/14016225/INSTRUCT; and the European Union's Horizon Europe program under Grant Agreement No. 101148870 (ContTestCPS); the European Union's Horizon 2020 research and innovation programme under Grants No. 871259 (ADMORPH project); the ELLIIT Strategic Research Area; and from the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Max Nyberg Carlsson and Martina Maggio are members of the ELLIIT Strategic Research Area at Lund University. 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.

References

[1] Claudio Mandrioli, Max Nyberg Carlsson, and Martina Maggio. 2023. Testing Abstractions for Cyber-Physical Control Systems. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 18 (nov 2023), 32 pages. https://doi.org/10.1145/3617170