Control-Theoretical Perspective on Feedback-Based Systems Testing

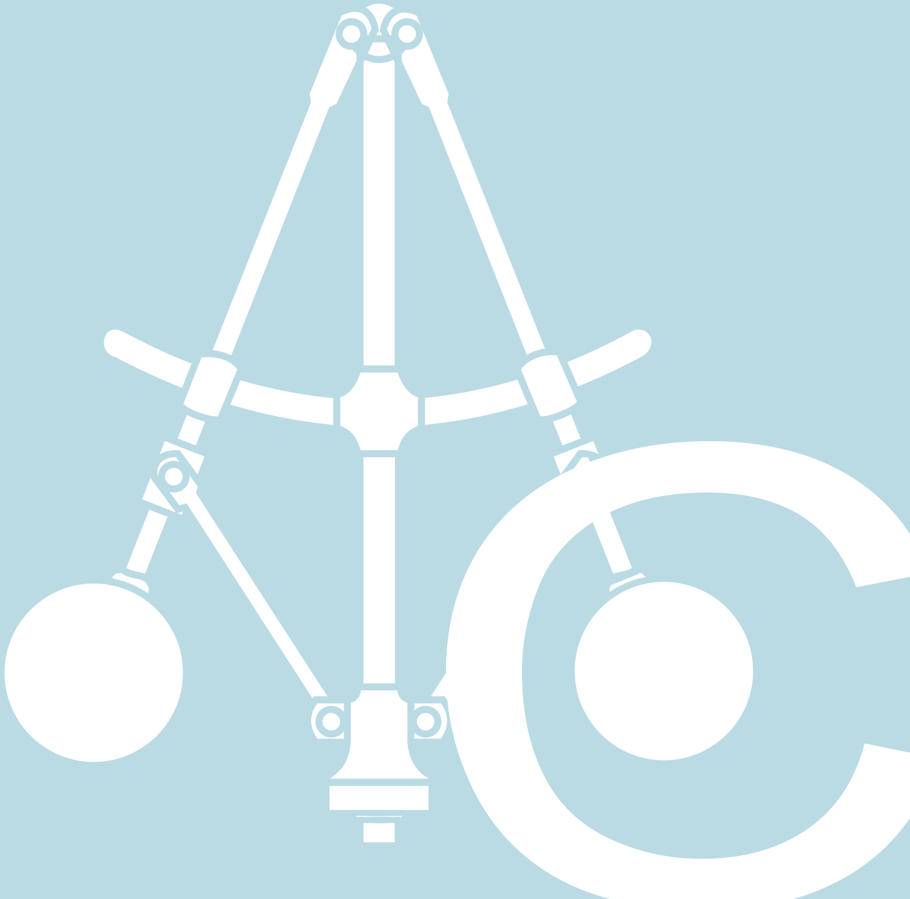
CLAUDIO MANDRIOLI
DEPARTMENT OF AUTOMATIC CONTROL | LUND UNIVERSITY

Printed by Media-Tryck, Lund 2021 NORDIC SWAN ECOLABEL 3041 0903

LUND
UNIVERSITY

Department of Automatic Control
P.O. Box 118, 221 00 Lund, Sweden
www.control.lth.se

PhD Thesis TFRT-1140
ISBN 978-91-8039-433-8
ISSN 0280–5316



Control-Theoretical Perspective on Feedback-Based Systems Testing

Claudio Mandrioli

LUND UNIVERSITY

Department of Automatic Control

PhD Thesis TFRT-1140
ISBN 978-91-8039-433-8 (print)
ISBN 978-91-8039-434-5 (web)
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Dedicated to Maria Cristina, Dino, Leonardo, and Laura who led me by example in my journey till here.

Abstract

Self-Adaptive Systems (SAS) and Cyber-Physical Systems (CPS) have received significant attention in recent computer engineering research. This is due to their ability to improve the level of autonomy of engineering artefacts. In both cases, this autonomy increase is achieved through *feedback*. Feedback is the iteration of sensing and actuation to respectively acquire knowledge about the current state of said artefacts and steer them toward a desired state or behaviour. In this thesis we discuss the challenges that the introduction of feedback poses on the verification and validation process for such systems, more specifically, on their testing. We highlight three types of new challenges with respect to traditional software testing: alteration of testing input and output definition, and intertwining of components with different nature. Said challenges affect the ways we can define different elements of the testing process: coverage criteria, testing set-ups, test-case generation strategies, and oracles in the testing process. This thesis consists of a collection of three papers and contributes to the definition of each of the mentioned testing elements. In terms of coverage criteria for SAS, Paper I proposes the casting of the testing problem, to a semi-infinite optimisation problem. This allows to leverage the Scenario Theory from the field of robust control, and provide a worst-case probabilistic bound on a given performance metric of the system under test. For what concerns the definition of testing set-ups for control-based CPS, Paper II investigates the implications of the use of different abstractions (i.e., the use of implemented or emulated components) on the significance of the testing. The paper provides evidence that confutes the common assumption present in previous literature on the existence of a hierarchy among commonly used testing set-ups. Finally, regarding the test-case generation and oracle definition, Paper III defines the problem of stress testing control-based CPS software. We contribute to the generation and identification of stress test cases for such software by proposing a novel test case parametrisation. Leveraging the proposed parametrisation we define metamorphic relations on the expected behaviour of the system under test. We use said relations for the development of stress testing approach and sanity checks on the testing results.

Acknowledgements

First and foremost, I'll never be able to thank enough my supervisor Martina. You have been a great and reliable guide for me through this journey under every aspect, from the professional to the personal ones. I'm truly grateful for everything you have done for me.

Secondly, I'd like to thank everyone at the Automatic Control department for making my time there such a pleasant experience. Eva, Mika, Cecilia, and Monika for running the department in such a smooth way. A special thank is for Eva for always sharing a smile and caring words. Thank you to all the colleagues and friends with whom I shared the office with and that made heading to work something to look forward to in the morning. A special mention goes to Richard for having always interesting nuances of mathematics to share and for the support with the provocative jokes with my supervisor, to Marcus for the crazy trips, delicious dinners, and deep conversations, to Ylva for being always positive and truly caring, and to Nils for all the work together and the valuable friendship. I cannot mention everyone here but I am grateful to all of the members of the department that I have had the luck to share my time there with, inside and outside of the university buildings.

I'm thankful to all the people I have had the pleasure to share the challenge of writing a paper. A special thank goes to Anton for being a great co-supervisor with always a positive attitude, interesting intuitions and insights, and despite putting banana on pizza. I'm grateful to my current and future collaborators from University of Luxembourg: Seung, Domenico and Lionel. Thank you for all that you have taught me. I am looking forward to continue the work we started together.

I cannot miss mentioning the friends outside of the office that I have shared the PhD journey with. Special mention goes to Ana, you have been great support in my first months in Lund helping me through my first impact with the PhD student life and with Sweden. To David, thank you for all the office hours shared at home during covid times, as well as the discussions about science, philosophy and politics. To Gabrielle, my both cycling and PhD friend, thanks for making many bike rides also an opportunity to share the ups and downs of this experience.

Last but not least, I am deeply grateful to my family that, despite the distance, has been of great support throughout these five years. Especially, to Dino, for always having clear and accurate views on the workings of the academic world, and to Daniele, for sharing with me this experience in distance.

Contents

1

Introduction

In recent decades, Cyber-Physical Systems (CPS) and Self-Adaptive Systems (SAS) have been two central topics in computer engineering research. CPS are systems that pair, through sensors and actuators, a physical and a computing component [Lee, 2015]. The coupling of the two components makes their analysis and design dependent on one another. Prominent examples of CPS are autonomous cars and drones, where the cyber component enables the autonomous motion of the system in an uncertain physical environment. SAS are software systems that monitor the changing environment in which they operate and modify their runtime behaviour accordingly [Diniz and Rinard, 1997]. This allows them to achieve system objectives independently of the operating conditions. Examples of such systems are scalable cloud services that adapt to the time-varying computational demand, or smart camera networks that change their behaviour depending on the video content.

At the root of the success of both SAS and CPS there is the ability to enhance and create systems that operate without human intervention, while optimising their resource consumption and improving their performance. An autonomous car should drive from the starting point to a desired location, by independently identifying the optimal route and adapting it to the changing surroundings, like roadworks and accidents. The car should also optimise its fuel consumption, and minimise the risk of accidents, especially when compared to human-driven cars. An autonomous cloud service should allocate or release virtual machines without human intervention, according to the requests received by its users. It should also minimise the number of running machines so that energy and cooling needs of the servers are minimised.

Systems like the ones mentioned above are more and more present in our daily life and responsible for increasingly critical tasks [Wu et al., 2017]. Therefore, assessing their correct functioning is of prime importance. This thesis contributes to the assessment of the behaviour of CPS and SAS, and in particular to the methodologies available for their *testing*. This chapter presents the fundamental concepts and challenges needed to frame the contribution of this work. To cast the explained concepts into a more practical and relevant use case, we use an autonomous car, an application that appears in both CPS literature [Kim et al., 2013] and SAS literature [Weyns, 2017].

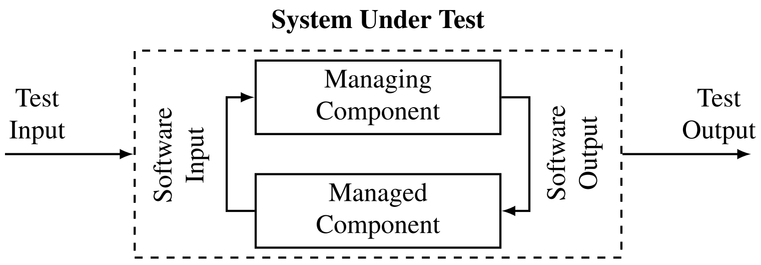


Figure 1.1 Testing of a system with feedback. The figure highlights that the overlap between testing inputs and outputs and software inputs and outputs characteristic of traditional software testing is no longer valid.

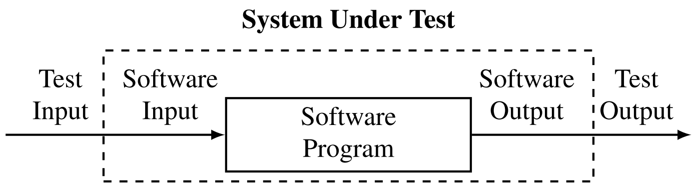


Figure 1.2 Traditional view of software testing where the testing inputs and the outputs are also the inputs and outputs of the software under test.

1.1 Fundamentals of CPS, SAS and their Testing

At the core of both CPS and SAS there is the concept of *feedback*. Feedback is a loop of cause-effect phenomena in which a managing component iteratively senses and actuates a managed component (as shown inside the dashed box in Figure 1.1). We say that the managing component, by measuring and actuating the managed component, *closes* the feedback loop around it. For example, an autonomous car detects the presence and movement of obstacles and objects around itself, analyses the potential risk of collision and plans its own trajectory accordingly. A change in the planned trajectory will affect the relative position and motion of obstacles and objects, hence closing the cause-effect loop. Being such a general concept, feedback has found application in different fields, ranging from the ones addressed in this thesis, the Monitor-Analyse-Plan-Execute loop for SAS and the sense-actuate loop in CPS, to others like the Observe-Orient-Decide-Act, loop for strategic decision making [Boyd, 1995], and the carbon cycle in climate modelling [Hansen et al., 1984].

In the context of feedback-based systems, *control theory*, also known as feedback-systems theory [Åstrom and Murray, 2008], offers a set of powerful tools for their analysis and design. Traditionally, control theory has been used mainly for the design of industrial controllers (e.g., programmable logic controllers) and the low level physical interface of CPS (a more detailed discussion on the role of control in CPS can be found in Section 1 of Paper III).However, recent research has proven it's applicability also to the design of adaptation strategies for SAS, sparking a series of works that cross the bridge between the two fields [Filieri et al., 2011; Filieri et al., 2017].

In the case of CPS and SAS, the decision-making process between sensing/monitoring and actuation (i.e., the closing of the feedback loop) is implemented with the use of software. Like for any other software project, it is important to evaluate that the software implementing the feedback in a SAS or a CPS makes the system compliant with the requirements and specifications. This activity is commonly known as the Verification and Validation process (V&V) and usually comprises of different activities—e.g. testing, code reviews, model checking, symbolic execution. In the case of software, testing is arguably the most used V&V activity by practitioners [Briand et al., 2016; Garca et al., 2020]. Testing is usually a dynamic V&V process, meaning that it leverages the system execution. This is opposed to static approaches like formal methods and code reviews that instead analyse the artefact [Garoche, 2019].[1] During the testing process, the system-under-test (SUT) is executed and fed with the chosen test input values. At the same time, the test output (i.e. the set of quantities needed to check the fulfilment of requirements and specifications) is measured, as shown in Figures 1.1 and 1.2.

The testing of a software that implements a feedback loop presents novel challenges that make traditional testing techniques inadequate. This has been evidenced both in the SAS research community [Siqueira et al., 2016] and the CPS research community [Briand et al., 2016]. The cause-effect loop introduced by feedback makes testing of CPS and SAS software different from the traditional testing of software. On an high-level, the input-output framework of traditional software testing finds limited applicability. More specifically, traditional software testing identifies the testing inputs with the software inputs and the testing outputs with the software outputs, as exemplified in Figure 1.2. In the case of systems with feedback, the definition of test inputs and outputs changes. As an example, if we want to test the ability of an autonomous car to avoid a moving obstacle, the test input is the movement of the obstacle. The obstacle movement is a quantity that lives in the managed component of the feedback system, and is different from the input to the software executed on the car. Similarly, the test output is the observation of whether the car has hit the obstacle or not. To be evaluated, also the output requires to reason about the physical (or managed) part of the SUT.

Another set of challenges that arises from the presence of a feedback loop in the SUT is the different nature that usually characterises the managing and managed components. For example, in an autonomous car, the physical car is the managed system and its behaviour is usually described with equations from physical modelling. The managing part is instead the ensemble of the microcontrollers and the

[1] A discussion on the difference between static and dynamic testing is reported in Section 2.1. In this work we focus on dynamic testing, i.e. on testing that includes the execution of the system under test.

software that they execute. Such difference in the nature of the components generally implies that they are developed by different engineers and that said engineers possibly leverage different modelling formalisms and methodologies. The V&V process for such systems has therefore to take into account the different responsibilities of the engineers involved with the development process. For example, a failure of an autonomous car could be equivalently due to a fault in the design of the planning algorithm, as well as in its software implementation, or in the design of the car geometries. Due to the causality loop introduced by the feedback, when the failure occurs, each of the three components is affected and ruling out which is the faulty one can be difficult.[2]

In this context, this thesis sets out to leverage tools from control theory for the effective testing of systems with feedback. Previous work has used control theory during the design of such systems. This is the first work that proposes its use for the testing. More specifically, the thesis leverages control theoretical tools for: (i) the definition of coverage criteria that are general to (independent of) the models of the managing and managed components in SAS system, (ii) how to design testing setups to distinguish the responsibilities of the different engineers during the testing of control-based CPS systems, and (iii) how to generate test cases for CPS systems developed with the use of control theory.

1.2 Challenges in Testing Feedback-Based Systems

As discussed above, the introduction of feedback poses different challenges for the testing process. In this section we discuss in further detail and provide examples for the ones addressed in this thesis. More specifically, we discuss:

- the alteration of the definition of test input,
- the alteration of the definition of test output, and
- the intertwining of components of different nature.

Test Input Alteration

Figure 1.1 is used to exemplify that the test input to the SUT, for systems with feedback, are not the inputs to the software. In fact, when feedback is used, the software receives two types of inputs: some that are the measurements from the managed component (i.e. the ones used to close the feedback loop) and others that are exogenous to the loop. In an autonomous car, examples of measurements from the managed component are the lidar readings or images from cameras. Examples of signals exogenous to the feedback loop can be user commands like the desired destination.

[2] The motivating example of Paper II reports an instance of such problem.

The two sets of inputs to the software are fundamentally different from the point of view of testing. The exogenous inputs can usually be arbitrarily set: for example the testers can define any desired sequence of desired destinations for an autonomous car that they believe can provide insight on the SUT. Conversely, the measurements are affected by the outputs of the software itself through the loop with the managed component and therefore they cannot be arbitrarily set [Gaaloul et al., 2020]. For example, in an autonomous car, the testers cannot arbitrarily change the speed measurements from the encoders mounted on the wheels. If the software opens the throttle, then the car accelerates, and the measured speed has to increase accordingly: the measurement cannot follow an arbitrary profile.[3]

Furthermore, those measurement do not depend only on the software output and are usually also affected by external conditions that affect the managed component. For example, the road conditions will affect how much speed a car will gain when accelerating and hence also the encoder readings. As a consequence, said external conditions have to be considered part of the test inputs for the SUT.

To summarise, the presence of feedback alters the definition of the test inputs for a software implementing feedback in two ways. Firstly the measurements from the managed system have to be representative of the closed loop interaction, and secondly, the external phenomena that can affect the managed component are now also part of the input set of the software. In general terms, we can say that *the test inputs for a system with feedback are all signals exogenous to the feedback loop and that affect it*. Said inputs include both the exogenous signals affecting the managing component and the ones affecting the managed component. Therefore, this alteration changes the nature of the test inputs with respect to traditional software testing and consequently changes how test cases can be generated and how the input space can be explored. Said in other words, feedback calls for novel test case generation strategies and input coverage criteria that account for the different nature of the managing and managed components.

Test Output Alteration

Here, with test outputs, we refer to the quantities that we measure when we perform a test in order to evaluate if the test is passed or not. In the case of systems with feedback, the requirements usually concern quantities that belong to the managed component. In fact, feedback is in the first place introduced to improve the behaviour of the managed component. For example, when evaluating if an autonomous car has reached its destination, we do not evaluate the output or final state of the software, rather we evaluate the final position of the vehicle in the physical space.

This implies that the performance of the software is not quantified on the base of its own output or behaviour (as assumed in traditional software testing) rather on

[3] Apparently, it can be interesting to test the system also in presence of sensor faults that alter the feedback signal. However, in that case, the feedback loop is opened (since the measurement does not depend any longer on the behaviour of the managed part) and the feedback implemented in the managing component cannot be properly tested.

how it influences the behaviour of the managed component. Therefore, *the set of the possible outcomes of the test, is defined in the domain of the behaviours of the managed component*, and not in the domain of the output values of the software. For example, when testing an autonomous car, we want to define the different possible obstacles that can appear and evaluate the software's ability to make the car avoid them rather than the raw throttle and breaking profiles that it generates. In practice, this alteration of the test output in systems with feedback calls for the definition of novel output coverage criteria and test oracles.

Intertwining of Components with Different Nature

For each component in the loop, feedback creates a connection between its own output and input. Therefore, the component's execution and testing requires, at the very least, an emulation of the rest of the loop. The objective of this emulation is to recreate said relation between the component's own output and input. This need to recreate the loop around the component implies that *single components in the loop cannot be tested in isolation*. For example, we cannot test the software that performs the on-line path-planning in an autonomous car without emulating how its commands affect the movement of the car. In fact, said commands are supposed to affect the car's position and therefore the inputs to the path planning software itself. Without iteratively feeding back this signal to the software, its execution is impossible.

In systems with feedback, the components' intertwining is such that, if one component is faulty, all of the other components in the loop are likely to also misbehave. If one single component is faulty, then its own output is affected: as a consequence the other components in the loop will observe a relation between their own input and output that is different from the one expected and will also show unexpected behaviour. Take as an example the case in which the software that steers the front wheels in an autonomous car is faulty and under-steers the vehicle (i.e. steers the wheels less than requested). The on-line path-planning algorithm will expect higher steering action and therefore generate a trajectory that is unsuitable for the car to follow. Because of this, the measurements do not reflect what it expected and the path-planning algorithm will change the trajectory. As a consequence, the resulting trajectory will possibly not be the optimal or expected one. In this context, it is hard to distinguish if the cause of the incorrect path is in the path-planning algorithm or in the software that controls the steering (or any other element in the loop, like for example the software that estimates the position of the car).

As it can be also noted from the example, the domains of the engineers involved are very different and they possibly use different classes of models. In the just mentioned example, the path-planning algorithm is possibly designed by a robotics engineer, the software that is implementing it might be written by a team of software developers (in the context of a larger codebase), and the firmware controlling the steering wheel might have been written by embedded systems engineers. Each of those fields has very different models to describe their part in the system: as examples, the robotics engineer possibly uses differential equations, the software team uses UML diagrams and the embedded systems engineers might use state machines. Developing a testing strategy that enables the distinction between the responsibilities of the different domains and covers their interactions is therefore challenging.

To summarise, *feedback systems are limitedly tested at component level and require significant testing in integrated environments* (both with simulation models and without). Furthermore, the components in the loop likely belong to different domains. Therefore, there is need for the systematic definition of testing setups and environments that allow the distinction of the different responsibilities of the different engineers involved.

Outline

This thesis is written as a collection of papers and consists of three context chapters and three papers. Of the three chapters, this introduction describes the basic concepts needed to illustrate the contribution of the work. Furthermore it introduces some of the high-level challenges that they pose. In Chapter 2 we introduce a brief historical perspective on the fields of control and software engineering. We then introduce the concepts from both fields needed for the rest of the thesis and the relevant related work. Finally in Chapter 3 we illustrate the contribution of each paper to the field. This latter chapter also reports the author contribution statement of this thesis' author.

2

Background

This chapter opens with a brief historical overview on the origin of the fields of control theory, software engineering, software testing, self-adaptive systems (SAS), and cyber-physical systems (CPS). The overview is based on the events reported in the timeline in Figure 2.1. The rest of the chapter reports the technical background and related work relevant for this thesis.

Brief Historical Overview

The origin of control theory is commonly accepted to correspond to the seminal paper written by the physicist James Clerk Maxwell in 1868 [Maxwell, 2011]. In said paper, Maxwell proposed a mathematical analysis of the the centrifugal governor invented by James Watt. The purpose of this centrifugal governor was to control the speed of a steam engine. The origin of software engineering comes instead around one century later. While there is not common agreement on the exact event

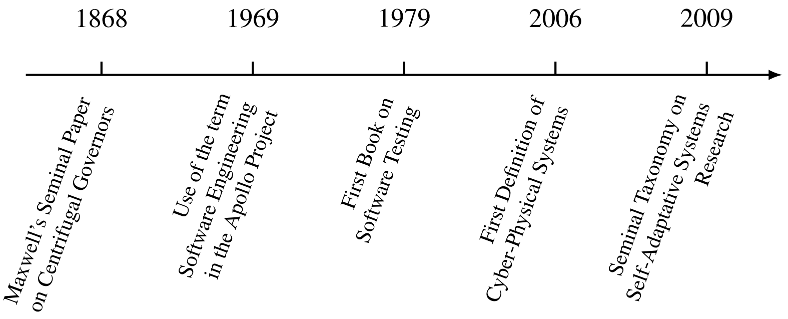

1868
1969
1979
2006
2009
Maxwell's Seminal Paper on Centrifugal Governors
Use of the term Software Engineering in the Apollo Project
First Book on Software Testing
First Definition of Cyber-Physical Systems
Seminal Taxonomy on Self-Adaptative Systems Research

Figure 2.1 This figure shows the timeline of the seminal events mentioned in the historical overview.

or date, one of the most cited episodes is arguably the use of the term by Margaret Hamilton in the 1960' during the Apollo project [*Message from the 50 Years of Software Engineering Chairs* 2018]. In the Apollo project, Margaret Hamilton was the lead developer of the spacecraft control software. It is curious to note that this first project acknowledged to be a software engineering one, was the implementation of a control system. However, this can be seen more as a coincidence and, exception made for few works, the fields of software engineering and control engineering are distinct.

Soon after the origin of the field of software engineering, the quickly growing complexity of software artefacts demanded for more advanced and systematic V&V tools. Accordingly, around one decade later, the software testing subfield established itself, and one of the first books exclusively dedicated to software testing was published in 1979 [Myers, 1979].

In the following decades, software systems continued to grow in size and complexity. This growth eventually resulted in a loss of predictability in their behaviour. To compensate for this predictability loss, researchers proposed to apply feedback to software systems. In this way, a software system can monitor itself and adjust its own behaviour in order to best meet its requirements. The application of feedback to software systems gave origin to the field of SAS. A seminal paper providing a taxonomy of the SAS field was written in 2009 [Salehie and Tahvildari, 2009]. However, more than a decade before, the paths of control theory and software engineering had already crossed again, with the first works that apply control theory in the context SAS being written around 1997 [Diniz and Rinard, 1997].

In the same years, computational devices had become more pervasive and available, thus increasing their connection with the physical world. For some software systems, the design of the software became tightly intertwined with the properties of the physical part that it has to operate with. This intertwining gave origin to the field of CPS, where the design of a software and physical components cannot be performed independently [Rawung and Putrada, 2014]. The first use of the term CPS is attributed to Helen Gill in the early 2000 [Gill, 2005].

As discussed in Chapter 1, both SAS and CPS are fundamentally characterized by a feedback loop. In the former, feedback appears in the form of software monitoring itself, while in the latter it appears in the form of the software interacting with a physical component. Furthermore, the previous chapter discussed how the presence of feedback poses new challenges in the testing of such systems. This thesis leverages models and tools from the field of control engineering to address said challenges and enhance the testing software systems that include feedback loops (i.e., SAS and CPS).

In the remainder of this chapter, we include the software testing and control engineering backgrounds that are relevant for this thesis. From the software testing side we want to define the problem and the type of the solution. From the control engineering side, we need to build the language needed to describe the SUT and its development, as well as the associated tools leveraged in this thesis.

2.1 Software Testing Background

This section introduces the relevant background on software testing. The first subsection introduces the testing fundamentals necessary to frame the contributions of this thesis. The second subsection introduces instead two advanced testing topics relevant for this thesis: Fuzz Testing (connected to Paper I) and Metamorphic Testing (connected to Paper III). Afterwards, in the third and forth sections, we discuss the related literature specific to the testing of SAS and CPS.

Software Testing Fundamentals

The literature proposes different definitions of software testing. A seminal definition of software testing attributed to Glenford J. Myers [Myers, 1979]: "*Software testing is the process of running a program with the intention of finding errors.*". In this definition there is an explicit reference to the execution of the SUT; nowadays, this is usually referred to as dynamic testing. In fact, other software testing definitions differ on this aspect, like the ISO/IEC/IEEE 29119 standard ["ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:General concepts" 2022] that defines testing as the "*set of activities conducted to facilitate discovery and evaluation of properties of the software under test.*" This latter definition includes both static and dynamic testing, where static testing includes other V&V techniques that do not involve the execution of the program. Examples of static testing activities are code reviews and static analysis. In this thesis we discuss solely dynamic testing.

On the high level, four elements are needed to systematically test a system:

- A test case generation strategy: which tests should we run?
- An oracle: is a tests output acceptable or not?
- A coverage or stopping criterion: when should we stop testing?
- A testing set-up: where and how should we run the test?

We now briefly define and discuss the role of each of them.

Test Case Generation. Test case generation is the definition of which inputs should be fed to the SUT when executing it. Ideally, we would like to test a system for any possible input and exhaustively explore its behaviours. This is apparently impossible for any non-trivial program and some strategic choice in the selection of the inputs has to be made.

Test case generation strategies can be divided in two classes, white-box and black-box testing, depending on the type of information that they leverage. Pure white-box testing leverages knowledge of the internal working of the software, while pure black-box testing leverages only information external to the software (e.g. its interface and requirements). An example of white-box testing is the generation of test cases according to the control flow of the program under test. If a program is supposed to identify leap years, and we know that it takes different execution paths for odd and even years, then such test case generation strategy will provide one test case with an even year input and one with an odd year input. On the contrary, an example of black box testing is to define a partitioning of the possible outputs on the base of the requirements. The partition can then be covered with test cases for each subset of the outputs. Continuing with the leap year detection program, the possible outcomes are that a year is leap or that it is not. Accordingly, this test case generation approach would generate two test cases, one that is a leap year and one that is not. Apparently, testing techniques can also leverage both types of information (internal and external to the SUT) and are referred to as grey-box testing.

Test Oracle. The oracle is the method to evaluate the test outcome, i.e. to state whether a test is passed or failed. Ideally, we would like an artefact that always provides the correct output of the program so that we could asses the the SUT output by direct comparison. Apparently, this is a paradox, as the oracle (i.e. the artefact always providing the correct output) becomes by definition a correct implementation of the desired program, hence defying the need of implementing the program in the first place. Therefore, the approach is generally to use the requirements to define expected properties of the output that can be verified. For example, considering again the leap year program, a property like "no odd year is a leap year" can be used as an oracle: if an odd year is identified as leap then we have detected a program failure.

The immediate consequence of the use of said properties, is that the oracle will not be always correct: meaning that it could pass tests that should be failed or vice-versa fail tests that should pass. Accordingly, common properties that are discussed about oracles are soundness and completeness: an oracle is said to be sound when it fails all tests that should fail (but also some that should pass). An oracle is instead said to be complete if it passes all tests that should pass (but also some that should fail). Soundness is a much harder property to achieve. For example, the oracle for the leap-year program above is only complete, as it will pass every test with even years (while some might still be incorrect).

Coverage or Stopping Criterion. The coverage or stopping criterion is the method that the engineers use to decide when they have tested enough and do not need to execute more test cases. It can also be used to evaluate the quality of a test suite, by quantifying how much it covers the possible behaviours of the SUT (also enabling the comparison of different test suites). As mentioned for the test case generation, we would ideally test a system for all its possible inputs and hence achieve complete coverage of its possible behaviours. Since this is not possible, and we have to resort to a finite number of test cases, not only we have to define how to generate them, but we also have to decide when we have generated and/or executed enough. In this context, a good coverage criterion exposes the trade-off between the number of tests and the exhaustiveness of the exploration of the SUT behaviours.

Examples of common coverage criteria are:

- statement coverage (are all the lines of code executed at least once),
- branch coverage (are all the branches of the conditional statements taken at least once), and
- condition coverage (are the different values of the boolean variables in the conditional statements evaluated to all possible values at least once).

When picking one of those criteria, the engineers can quantify an associated metric (e.g. the percentage of lines of code that are executed by the tests at least once), and set a target value that they want to achieve. Once the target value is reached they consider themselves satisfied with the testing and can stop it.

Testing Set-Up. The testing setup consists of the external conditions in which the SUT is executed for testing. It is important that said conditions enable the exposition of the properties of interest in the SUT—i.e. the ones defined in the requirements. Which external conditions are relevant depends significantly on the specific application: for example, for a smartphone app, the operative system within which the app is tested is relevant. Differently, in the case of a floating point arithmetic library for embedded devices, the relevant conditions include the hardware architecture. Ideally, we would like the testing set-up to be equivalent to the environment in which the software will be executed once it is deployed. This is not always possible for different reasons: for example, there might be too many possible deployment environments (e.g., all of the smartphone models for the app), or executing in the deployment environment might be very expensive in time and resources (e.g., if the floating point library is going to be deployed on an aircraft).

These limitations in the representativeness of the testing environment introduce trade-offs in the design of the set-ups. On the one side, said trade-offs have the coverage of the possible deployment environments and their representativeness. On the other side, they have the cost of implementing and maintaining many and detailed environments to execute the tests. For example, for testing the mobile app the testers will have to select how many and which smartphones to use. Or potentially they could use an hardware emulator, that might not behave exactly as the target devices but has a lower execution cost.

Advanced Software Testing Techniques

Thanks to its results,random sampling of the system inputs has recently gathered significant attention as test case generation approach [Chen et al., 2010; Arcuri and Briand, 2011; Holler et al., 2012]. This approach has gained popularity under the name of *Fuzz Testing* [Zeller et al., 2019; Yatoh et al., 2015; Tramontana et al., 2019;

Böhme, 2019] and has found most of its applicability in security and vulnerability testing [Holler et al., 2012]. Some recent reviews on the vast literature on Fuzz testing can be found in [Yun et al., 2022; Lan and Sun, 2021; Shen et al., 2021].

Metamorphic Testing was first proposed in 1998 to enhance test case generation and support the detection of faults [Chen et al., 1998]. This enhancement is achieved by leveraging properties (called metamorphic relations) defined over two or more test cases—e.g. if the value of a test input is increased then also the output value should. This is in contrast with traditional testing that treats the different tests individually. A recent literature review can be found in [Chen et al., 2018]. Notably, we report one application of Metamorphic Testing to control software testing [Chen et al., 2011] and a recent industry paper on the synthesis of metamorphic relations for CPS [Ayerdi et al., 2021]

Testing of Self-Adaptive Systems

A very recent survey on the whole field of SAS can be found in [Wong et al., 2022]. The problem of testing an adaptive software – in some cases also called *context-aware* software [Wang et al., 2014; Micskei et al., 2012] – is not a new challenge for the software testing community [Siqueira et al., 2016; Oliveira Neves et al., 2018]. The software engineering literature that addresses the testing of self adaptive software includes both design-time and run-time approaches (with the latter having received significantly more attention in the literature [Wong et al., 2022]).

The design-time approaches to testing self-adaptive software include SIT [Qin et al., 2016] and TestDAS [Santos, 2017]. SIT [Qin et al., 2016] proposes a test case generation technique for self-adaptive applications. The sampling of the input space is based on an interactive model of the application that is being tested. TestDAS [Santos, 2017] focuses instead on triggering the adaptations during the execution of the test cases. It leverages models of the software behaviour that are defined in advance by the programmer.

The literature on software testing includes several efforts to develop run-time testing methodologies for adaptive software [Cheng et al., 2014; Hänsel et al., 2015; Reichstaller and Knapp, 2018; Wong et al., 2022]. Generally speaking, there is a need to develop models for verification and validation at run-time [Cheng et al., 2014]. This need is caused by the ever-changing nature of the environment the adaptive software operates in.

In a SAS, not only the environment changes, but also the software itself does. Complementarily to the works above (that focus on the environmental changes), in works like [Silva and Lemos, 2011; Fredericks and Cheng, 2015] the authors develop a testing methodology that adapts the test cases to the architectural reconfigurations in the SUT. A similar problem is addressed with a run-time analysis approach in [Salama and Bahsoon, 2017].

Context-aware software is close to self-adaptive software, and there is a significant amount of work addressing the problem of testing context-aware applications [Wang et al., 2014; Yu et al., 2014; Mehmood et al., 2018; Micskei et al., 2012]. The self-adaptive (or context-aware) software observes the execution environment and selects actions to be performed based on the result of the observation phase. The research effort for context-aware software goes in the direction of generating test cases that trigger the context-aware software layer [Yu et al., 2014; Mehmood et al., 2018; Micskei et al., 2012]. In [Yu et al., 2014], automatically generated bigraphs are used to model the interactions between the environment and the software, and to generate the test cases. In [Mehmood et al., 2018] the authors propose a framework for automatically generating test cases with high-level test data.

Testing of Cyber-Physical Systems

Testing of CPS is a broad research field [Zhou et al., 2018; Abbaspour Asadollah et al., 2015]. In order to focus the scope on the relevant work for this thesis we limit this discussion to the testing of control-based CPS—i.e. the testing of CPS that are developed also with the use of control-theory [Balasubramaniam et al., 2020; Bradley and Bagheri, 2020; Broy et al., 2007]. The testing of control-based CPS appears in the software engineering literature in different ways. We divide the relevant literature in the following categories:

- model-testing and model-based testing,
- testing of embedded software,
- CPS falsification,
- application-specific control-software testing, and
- efforts from the control community.

Model-based approaches are well suited for control-based CPS [Nielsen et al., 2015]. In fact, software tools like Simulink have gained widespread industrial use [Menghi et al., 2019]. Following this drive, model-based testing (i.e. leveraging models to guide the testing campaign) and model-testing (i.e. the testing directly applied to executable models instead of the product) have received significant attention in the last decade [Briand et al., 2016]. Furthermore, recent work shows the complementarity of model checking and model testing for the verification of requirements [Nejati et al., 2019]. The research literature is focused on generation of test traces [Hänsel et al., 2011], or the use of models for the automatic generation of test cases with search algorithms [Matinnejad et al., 2014; Matinnejad et al., 2017; Marculescu et al., 2015], classification trees [Lamberg et al., 2004], system-identification based refinements [Menghi et al., 2019], or search algorithms [Aleti and Grunske, 2015; Hänsel et al., 2011; Ben Abdessalem et al., 2018].

The embedded software literature sees control-software as a prime application of computing in a resource constrained environment [Garousi et al., 2018]. This is due to the imposition of strict constraints on non-functional properties (e.g., execution timing). Accordingly, control software appears in several surveys and reviews on the testing of embedded software [Zander et al., 2011; Garousi et al., 2018; Banerjee et al., 2016]. Notably, model-based approaches also frequently appear in the embedded software literature [Zander et al., 2011; Banerjee et al., 2016].

The cited reviews extrapolate different challenges of embedded software testing. We report a couple: the unavailability of a user interface, and the close integration of software and hardware [Garousi et al., 2018]. Because of said challenges, there are works discussing the relevance and properties of different testing set-ups. Intuitively, when there is tight connection between software and hardware, the testing set-up has a significant impact on the effectiveness of the testing process. This led to the naming convention "X-in-the-loop" where "X" can be different system components (e.g. software, hardware, processor, system, process) and defines which components are included in the set-up. We note that this naming is used in slightly different ways across different works but with an overall consistency [Zander et al., 2011; Bringmann and Krämer, 2008; Lamberg et al., 2004]. We refer the interested reader to Section 3 of Paper II for a more detailed discussion on the definition of the "X-in-the-loop" testing set-ups.

It is interesting to note a number of application-specific publications on the specific topic of testing control software. Those concern mainly the fields of avionics [Peleska, 2002; White, 2001] and automotive [Bringmann and Krämer, 2008; Bringmann and Krämer, 2006; Bücs et al., 2016]. It is important to remark that a significant number of those are white papers. This latter consideration can be interpreted as a marker of the topic's industrial relevance and possibly the need for dedicated tools specific to control-based software testing.

CPS falsification (as the generation of test cases that falsify a requirement) is also an active research direction. In a recent work [Yamagata et al., 2021] the authors use deep reinforcement learning to perform robustness guided falsification on CPS. Interestingly, they note the importance of making the system internal dynamics available to the reinforcement learning algorithm, which highlights the possible benefits of leveraging control-design information during the testing. Differently, in [Plaku et al., 2009], the authors use a combination of model checking and path planning to invalidate linear -temporal-logic formulas. A similar problem is addressed with the use of rapid exploration of random trees in [Dreossi et al., 2015].

The control and robotics communities show growing (but still limited) awareness of the impact of the control-algorithms software implementation on their performance [Zimmer et al., 2015; Silano et al., 2018]. Zimmer et al. [Zimmer et al., 2015] discuss a case study on the consequences of implementation choices for the control performance. In robotics, [Silano et al., 2018] showcase the potential in exposing bugs of software-in-the-loop simulations (i.e. simulations that include the actual control software implementation) for the design of quadcopter controllers. We note that a similar work on the evaluation of such testing setup is found in the recent software testing literature [Timperley et al., 2018].

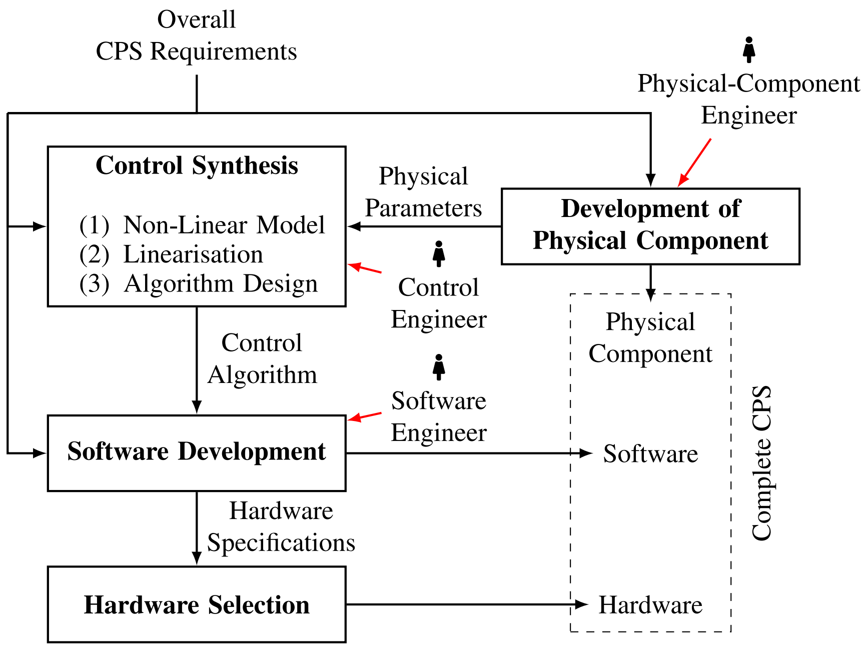


Figure 2.2 Graphical overview of development of control-based CPS. The overview is simplified to focus on the role of control engineers in the development of the interface between the cyber and physical components. In the figure, the boxes represent development steps and the human-shaped figures represent the different engineers involved. The black arrows represent artefacts or design information shared between the engineers, and the red arrows represent the execution of a development step by certain engineers. This figure is adapted from Figure 2 of Paper III.

To conclude, we note that, despite the significant number of publications on model-based testing and model-testing, none of the reported works explicitly leverages the fact that such systems are designed with the use of control theory. A partial exception are [Aleti and Grunske, 2015; Menghi et al., 2019]: in both cases, system identification (a field very close to, if not part of, control theory) is used to reduce the number of tests that have to be executed in order to find faults [Menghi et al., 2019] or to reduce the parameter definition effort required by genetic algorithms [Aleti and Grunske, 2015]. In said cases, the system identification tool is used as part of the testing process itself and not to gain insights on the SUT to be leveraged for its testing. In addition, we report a very recent work in the context of runtime verification that leverages system dynamics [Abbas and Bonakdarpour, 2022], where models of the CPS physical component based on differential equations (same type of models used at control design time) are used to either terminate the signal monitoring early or to skip parts of the signal.

2.2 Control Engineering Background

In this section we provide the control engineering background needed for this thesis. In the first part we introduce the development of control systems: this is needed to frame the contributions of Papers II and III. In the second part we introduce Scenario Theory: a tool from the field Robust Control that we leverage in Paper I. We complement the discussion on Scenario Theory with background on the statistical tools that, in the paper, we compare it against, namely: Monte Carlo methods and Extreme Value Theory.

Control Systems Design and Development

Figure 2.2 gives a graphical overview of the development of control-based CPS, from the definition of the requirements (top-left) to the combination of the physical component with the software and hardware that implement the cyber component (bottom-right, in the dashed box). This is a simplified overview that focuses on the role of the control engineers in the CPS development. The development of a CPS, like of any other engineering artefact, starts with the definition of the requirements.[1] In the case of CPS, requirements usually describe the desired behaviour of the physical component of the system, also in respect to commands provided by the user. For example, in an autonomous car, the requirements might include that the car is supposed to be able to safely drive to a requested destination, and to drive in a comfortable way for the passengers (e.g. without too high accelerations).

The requirements are then passed to the different engineers involved in the development of the CPS. In this context we identify mainly three types of engineers involved:

- the physical-component engineers that design the physical component of the CPS,
- the control engineers that design the control algorithms, and
- the software engineers that design the high-level controllers and the software implementation—i.e., the cyber component.

On an high level, engineers specific to the physical component (e.g., mechanical or aerospace engineers) provide information about the physical component to the control engineers. The latter uses said information to develop the control algorithms: as highlighted in the figure, this includes three main steps. In the first step, the control engineers use the information provided by the domain engineers to develop an equation-based model of the physical component, said equations are generally nonlinear and therefore difficult to analyse. Accordingly, the second step is to linearise

[1] A more detailed discussion of the control design process can be found in Section 2 of Paper III. Here we limit ourselves to mentioning the main steps.

the model in order to make it tractable. The linearisation introduces an approximation and limits the model's scope of validity.[2] On the other side, the linearisation allows the control engineers to use one of the several tools offered by control theory to develop the control algorithms. Such algorithms are then passed to the software engineers and are part of the software specification—said in other words, the software implements the control algorithms as well as other functionalities like sensors initialisations and sanity checks. The software engineers implement the CPS software and defines the performance required from the hardware. According to the needed hardware properties and performance, either an off-the-shelf platform can be selected or custom hardware might be needed (with the latter possibly requiring to involve electronics engineers for the platform design). Finally, software, hardware and physical component can be combined to obtain the complete CPS. For a more detailed discussion of the different control-based CPS components and their role we refer the interest reader to Section 3 of Paper II.

In the autonomous car example, the mechanical engineers design for example the car geometries, and the sizing of the brakes and engine. The control engineers use this information to design the control algorithms. In this context, control algorithms compute the commands sent to the engine and brakes so that the car can reach and maintain a desired velocity, as well as the commands sent to the steering wheel needed to drive in a desired direction. The software engineers implement the car's software that includes the control algorithms together with other functionalities like the user interface, path-planning algorithms and autonomy engagement/disengagement features. During this process, also the hardware of the car has to be designed, according, for example, to the needed computational power, memory and I/O interfaces.

In this context, we highlight that the different engineers use fundamentally different models and methodologies to develop their part of the system. For example, the mechanical engineers use finite-element methods to evaluate the mechanical properties of the car [J N Reddy, 2005]. The control engineers use high-level dynamics models to evaluate the reaction of the car to motor and brakes commands. The software engineers use UML diagrams to divide the software in modules.

Apparently, the overall satisfaction of the CPS requirements depends on each of the components functioning as required as well as on their correct interoperation. However, the multidisciplinary nature of this step makes it challenging to verify and validate them. Most notably, the development (and therefore the verification and validation) of the software is concern of both the control and software engineers. A more detailed discussion on the multidisciplinary nature of CPS can be found in Section 2 of Paper III.

[2] A more detailed discussion on models linearisation and, more in general, on the *design scope* of control algorithms can be found in Sections 1 and 2 of Paper III.

Scenario Theory and Tools from Statistics

In this section we first introduce Scenario Theory, a tool from the field of robust control. This theory is at the base of the contribution of Paper I. Afterwards, we introduce also the intuitions behind the tools from statistics that, in the paper, we compare against Scenario Theory: namely, Monte Carlo Sampling and Extreme Value Theory.

Scenario Theory. The *scenario approach* [Calafiore and Campi, 2006] was developed in the field of robust control [Francis and Khargonekar, 1995]. Robust control theory develops control algorithms that are able to perform consistently in presence of uncertainty—e.g., modelling errors and unknown parameters from the first step of control synthesis, Figure 2.2. For assumed bounds on the uncertainty, robust control aims at providing guarantees on the performance of the final system. Despite being developed in the field of robust control, scenario theory is applicable to a wider class of problems. In fact, it provides a method to solve the more general class of *semi-infinite convex optimisation problems*. Semi-infinite convex optimisation problems are a class of optimisation problems that appear frequently in robust control design, but are also found in other fields, such as decision making, finance, and management [Ramponi and Campi, 2018; Calafiore, 2013].

In order to describe the contribution of Scenario Theory we need to first introduce what is an optimization problem and what is a semi-infinite optimisation problem. In an optimisation problem we want to maximise or minimise a given quantity. This is done by selecting values for given variables, while in presence of constraints on the available choices. More rigorously, optimisation problems are defined by: (i) a cost function, (ii) one or more decision variables, and (iii) a set of constraints. The cost function is the quantity we would like to maximise or minimise, and it should be a function of the decision variables. The decision variables are the quantities that we can choose and change. The constraints are statements about the decision variables that we want our final solution to satisfy. An example of optimisation problem is the computation of the shortest path between two points in presence of obstacles to be avoided. The decision variables are the parameters of the path, the cost function is the total length of the path, and the constraints are the positions of the obstacles that have to be avoided.

When the number of decision variables of the optimisation problem is finite, but the number of constraints is infinite, we have a semi-infinite optimisation problem.[3] In the shortest-path example, infinite constraints would correspond to infinite obstacles. At first, accounting for infinite obstacles might not seem a practically relevant problem. However, infinite obstacles can be used to represent uncertainty in the position of a single obstacle. In presence of such uncertainty, we can choose to plan a path that avoid any possible position of the obstacle. If the set of said posi-

[3] Technically speaking, also an optimisation problem with finite constraints and infinite decision variables is a semi-infinite optimisation problem. However, in this work we are interested only in the class of problems with infinite constraints.

tions is infinite (e.g. if the set is dense), then we have a semi-infinite optimization problem. While solving a convex optimisation problem is in most cases feasible and there are well established tools that can quickly provide the solution, semi-infinite problems are harder to treat, and generally require approximate solutions [Boyd and Vandenberghe, 2004].

In this context, Scenario Theory [Calafiore and Campi, 2006] leverages a probabilistic framework to treat semi-infinite optimisation problems. The theory allows to use only for a finite number of randomly sampled constraints from the infinite set of possible ones. Then, it provides probabilistic guarantees on the generality of the solution, i.e., on the probability that the solution obtained with a finite number of constraints is valid for all of the infinite possible constraints. In the shortest-path example, its application would provide a result in the form of "with 99% probability the obtained path will avoid the obstacle", or, more rigorously, "the obtained path will avoid 99% of the infinite possible obstacles". Furthermore, the theory provides a confidence metric on the obtained result, i.e. a quantification of the probability that the obtained (probabilistic) bound is correct. In our example, this would be the probability that the statement above on the percentage of fulfilled infinite constrains is correct.

The fundamental result of Scenario Theory is that it allows to formally quantify and connect

- d the number of decision variables,
- N the number of randomly sampled constraints,
- ε the probability of satisfying all of the infinite constraints, and
- β the confidence.

An important feature of this result that we leverage in our work, is that *it holds irrespectively of the actual probability distribution of the constraints*. Said in other words, it does not require any assumption on the distribution underlying the uncertainty, and significantly simplifies its modelling. In the shortest-path example, this would allow to make no assumption on the probability distribution used to describe the possible positions of the obstacles. Section 4.3 of Paper I, provides a more detailed discussion on Scenario Theory and how it can be applied to solve SAS software testing problems.

Monte Carlo Sampling. Monte Carlo methods [Robert and Casella, 2005] use random sampling to numerically infer the value of quantities of interest. The idea is that we can reconstruct the properties of an unknown probability distribution by sampling it enough many times. For example, we might be interested in knowing how tall are the people living in a given country. To obtain full information about the height of people, we would ideally measure each individual and obtain the complete distribution. This is apparently practically impossible or at least very demanding.

The idea of Monte Carlo sampling would be instead to measure only some randomly chosen people and then use those data to infer properties like the average height of the whole population or the percentage of people that are higher than a given threshold.

The theoretical result leveraged by Monte Carlo methods is the Central Limit Theorem [Johnson, 2004]. The theorem discusses the mean $\bar{x}$ of the samples x_i from an arbitrary probability distribution X:

$$\bar{x} = {}^1\!/_n \textstyle\sum_{i=1}^{n} x_i. \tag{2.1}$$

The theorem states that, if one draws infinitely many samples from the random variable (i.e. $n \rightarrow \infty$), the distribution of their arithmetic mean asymptotically converges to a normal (gaussian) distribution, regardless of the original distribution that they are taken from. Mathematically, $\bar{x} \sim \mathcal{N}(E[X], {}^{\sigma^2}\!/_n)$, where $\mathcal{N}$ is the normal distribution, $E[X]$ is the expected value of the random variable X and σ^2 is the variance of X. On an intuitive level, this result could be expressed as: irrespectively of the nature of the underlying phenomenon, if we pick enough samples, the average behaviour observed in the random samples is representative of the actual phenomenon average behaviour. In fact, the expected value of $\bar{x}$ converges to the same expected value of X, and its variance ${}^{\sigma^2}\!/_n$ (i.e., the probability of the two being different) converges to zero. In our people's height example, we could say that, if we randomly pick enough people, the average of our samples can be considered representative of the average of the whole population.

Nowadays, Monte Carlo methods are employed in many different fields, from optimisation [Robert and Casella, 2010] to decision making [Jiménez-Martín et al., 2005]. In Section 3.2 of Paper I we provide more details on the theoretical foundations of the theory and especially on how Monte Carlo methods can be applied to the problem of software testing. However, we remark here two important features of the theory's application.

First, in order to apply Monte Carlo methods, it is of fundamental importance to not introduce biases during the sampling. In the population height case, we might for example use medical records to obtain our samples. This automatically excludes all the people that do are not registered in the medical database, hence making the sampling of the population biased towards those that have needed medical care. Depending on the specific reasons why people have needed medical care, this might or might not bias the final results.

Second, since the Central Limit Theorem discusses the average behaviour, Monte Carlo methods are well-suited to infer properties about the average behaviour of the phenomenon of interest. However, when it comes to atypical behaviours, i.e. when we are interested in properties of the tails of the distribution, Monte Carlo approaches fall short. In our population height example, if we are interested in knowing how tall is the highest person in the population, Monte Carlo methods are not well suited. In the context of software testing, this makes Monte Carlo approaches unsuitable to provide worst case performance bounds. To overcome said limitation, researchers from the field probability theory have developed Extreme Value Theory, which we discuss next.

Extreme Value Theory. Extreme Value Theory [Haan and Ferreira, 2010] studies the samples of a random variable around the tails of its distribution. This is opposed to Monte Carlo methods that study a variable's behaviour around its average. More specifically, Extreme Value Theory is used when we want to use observations to evaluate the maximum or minimum values that a variable can have, for example, if we want to analyse how tall is the tallest person in a country. The theory is nowadays widely adopted to study rare phenomena such as earthquakes, quantitative risks in finance, but also extreme events in engineering [Santinelli et al., 2014; Cazorla et al., 2013].

Extreme Value Theory is based on a result analogous to the Central Limit Theorem: the Fisher-Tippett-Gnedenko Theorem [Fisher, 1930]. This theorem discusses the distribution of the maximum $\hat{x}$ of a set of samples from a given distribution X:

$$\hat{x} = \max_i \{x_i\} = \max\{x_1, x_2, \dots, x_N\}. \tag{2.2}$$

It states that, if we draw sufficiently many samples form the random variable, the distribution of the maximum value among those samples is the Generalised Extreme Value Distribution [Haan and Ferreira, 2010].[4] However, this holds only if its distribution actually converges, which is generally not known a priori. This is an important element of difference from the Central Limit Theorem (that instead applies to any distribution with finite variance) because it restricts the general applicability of the Extreme Value Theory.

The application of Extreme Value Theory comprises of three main steps: (i) obtain a set of samples from the distribution that we want to study, (ii) extract a set of so-called maxima from the samples (i.e. the samples that we consider belonging to the tail of the distribution), and (iii) fit a Generalised Extreme Value Distribution to the maxima. The obtained distribution then describes the maximum value that could be observed in future realizations of the system. In this procedure, a critical step is the extraction of the maxima. There are different practices to extract the maxima from a dataset. The most common are: (i) the Block Maxima, and (ii) the Peaks Over Threshold. The former defines a partition on the dataset and extracts the maximum value from each subset, the latter takes all the values that exceed some predefined threshold. The difference between the two methods stems from the possibility of partitioning the dataset or not (for example if the performance data are acquired with different software releases). When the data is naturally partitioned into smaller sets, the block maxima methods is preferred. In the population height case Block Maxima would for example pick the tallest person from each region; Peaks Over Threshold would take all people above a used-defined height.

[4] The mathematical formula of the Generalised Extreme Value Distribution can be found in Paper 3, Section 3.2, Equation 3.

As mentioned above, differently to the Monte Carlo methods (which talk about the average behaviour of a system), Extreme Value Theory discusses the rare events in a system. For this reason, by definition, few tests carry information about such cases. The main consequence of this is that the convergence of the distribution of the extreme value to the Generalised Extreme Value Distribution is generally slow. This also limits the practical applicability of the theory. Section 3.2 of Paper I provides a more detailed discussion on Extreme Value Theory and how it can be applied to software testing. For both Monte Carlo methods and Extreme Value Theory, Section 4.1 of Paper I discusses the limitations of their application to software testing problems.

3

Control Perspective on the Testing of Systems with Feedback

In this chapter we describe the contributions of this thesis. We contextualise our contributions into the software testing problem using the four testing elements presented in Section 2.1: coverage criterion, test-case generation, testing set-up, and oracle. Figure 3.1 shows a graphical representation of the mapping between the different publications included in this thesis and the different elements of a testing process. Paper I uses the probabilistic confidence to evaluate the coverage of an SAS behaviour through randomized testing. Paper II investigates the impact of the testing set-ups on the significance of the testing results in control-based CPS. Paper III proposes a test-case parametrisation for control-based CPS which we leverage for

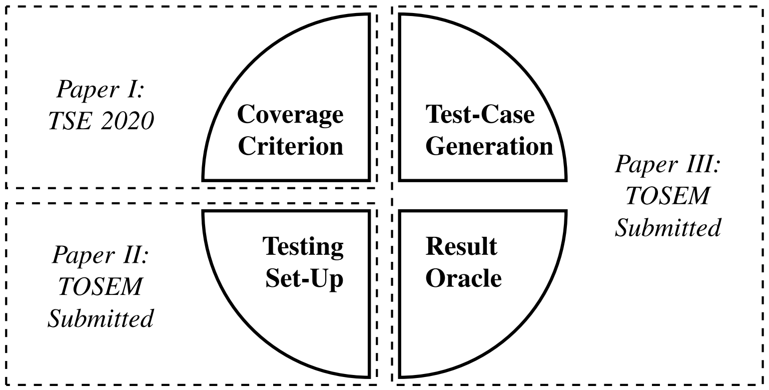


Figure 3.1 Mapping of the papers included in this thesis to the basic elements of a testing process. The 4 circular sectors represent the basic elements of a testing process, the dashed boxes show how they map to the papers of this thesis.

both test case generation and for the definition of expected properties of the SUT behaviour.

This chapter is divided in three sections each presenting the contribution of one included paper. Within the presentation of the contributions, we motivate them using the high-level challenges described in Section 1.2. Each section is closed with the author contribution statement for each paper. Finally, the chapter is closed with the references to the other papers published by the author of this thesis that haver not been included.

3.1 Coverage Criterion (Paper I)

C. Mandrioli and M. Maggio (2022). "Testing self-adaptive software with probabilistic guarantees on performance metrics: extended and comparative results". *IEEE Transactions on Software Engineering* **48**:9, pp. 3554–3572. DOI: `10.1109/TSE.2021.3101130`

Extends: C. Mandrioli and M. Maggio (2020). "Testing self-adaptive software with probabilistic guarantees on performance metrics". In: *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM. ISBN: 9781450370431. DOI: `10.1145/3368089.3409685`. ACM SIGSOFT Distinguished Paper Award.

Statement of Contribution. M. Maggio contributed the idea of applying the scenario theory to the testing of self-adaptive software. C. Mandrioli cast the idea into the specific application scenarios and implemented them either in simulation (for the tele-assistance service case study) or within real software (for the self-adaptive video encoder and the traffic routing platform). Both authors shared the writing of the manuscript.

Scientific Contribution. In Section 1.2 we discussed how the alteration of test inputs and outputs in systems with feedback calls for the definition of new input and output coverage criteria. In Paper I we address this problem in the context of the worst-case quantification for an SAS performance metric. Said in other words, we address the problem of quantifying the coverage of the SAS behaviours and of obtaining a performance bound that always holds for a given performance metric.

To address this challenge, we propose the use of a probabilistic approach to the testing problem—i.e., to randomly generate the test cases [Böhme, 2019; Dutta et al., 2019]. The probabilistic approach is beneficial in two ways: first, it allows the efficient exploration of large input space, and second, when the result is rigorously analysed, it can provide a quantification of its system coverage in the form of statistical confidence.

The randomised generation of test inputs enables the efficient input-space exploration thanks to its independence from the size and quantity of the uncertainty present in the software execution [Robert and Casella, 2005; Dutta et al., 2019].[1] However, the measured performance of the system (the test output) must now be treated as a random quantity, that takes values as a probability distribution rather than having a fixed value (as in traditional testing) [Böhme, 2019; Abu-Mostafa et al., 2012]. Accordingly, it requires a probabilistic evaluation. As thoroughly discussed in Section 4.1 of Paper I, in order to evaluate the performance probability distribution, traditional tools from statistics fall short because they require assumptions on the uncertainty that are not always fulfilled by SAS.

To address the quantification of the probabilistic guarantees on the worst-case performance and to quantify our testing coverage, *we cast our testing problem into a semi-infinite convex optimisation problem*. In this way, we can leverage Scenario Theory (introduced in Section 2.2) to solve the optimisation problem and obtain the probabilistic bound on the worst-case performance. At the same time, Scenario Theory allows us to quantify the testing coverage in the form of the probabilistic confidence on the obtained bound. The full explanation on how to cast the testing problem is reported in Section 4.2 of Paper I. Here we limit ourselves to remarking the powerful analogy between the *infinite* possible realizations of the SAS (because of the uncertainty that characterises such systems) and the *infinite* constraints of the optimisation problem. In the same way that Scenario Theory enables the use of only a finite set of random samples of the constraints in the optimisation problem, we leverage it to use only a finite set of random realizations of the SUT (i.e. a finite set of tests). Analogously to how we obtain a probabilistic coverage of all the infinite constraints for the optimisation problem, we obtain a probabilistic worst-case bound on the performance for the testing problem .

3.2 Testing Set-Up (Paper II)

C. Mandrioli, M. Nyberg Carlsson, and M. Maggio (2022a). "Testing abstractions for cyber-physical control systems". *Submitted to ACM Transactions on Software Engineering and Methodology*

Statement of Contribution. C. Mandrioli proposed the idea of investigating the properties of the different testing set-ups for control-based CPS upon literature review and discussions with M. Maggio. C. Mandrioli implemented the model-in-the-loop and hardware-in-the-loop set-ups. M. Nyberg Carlsson implemented the software-in-the-loop and process-in-the-loop set-ups during his master thesis.[2] C. Mandrioli had a leading role in writing the paper, with M. Maggio providing support. M. Nyberg Carlsson provided feedback and helped with the final proofreading.

[1] For example, a result like the convergence of the central limit theorem depends on the number of samples n and not on the size of the random variable sample space.

[2] M. Nyberg Carlsson performed part of his work in the context of an internship at Bitcraze, the company developing the Crazyflie drone used in the experiments.

Scientific Contribution. In Section 1.2 we discussed how the use of feedback creates intertwining between components of different nature. In Paper II we address this problem in the context of testing of control-based CPS (introduced in Section 2.2), where three elements (software, hardware and physics) of different nature are connected in closed-loop (Figure 2.2). More specifically, we address the problem of defining different testing set-ups that improve the distinction of the responsibilities of the different engineers in the development of the system.

While the idea of using different testing set-ups is a common industrial practice and is found in previous literature [Zander et al., 2011], this is the first paper that empirically compares their properties. In fact, previous works focus on individual set-ups. In a given testing set-up, some CPS components included as their final implementation and other components are instead emulated. In our work, we use the concept of *abstraction* to define the different testing set-ups. The emulation of a component implies the abstraction of its own behaviour as well as the abstraction of its interaction with the other components. The relevance of the testing abstractions comes from the fact that they implicitly carry *assumptions* on the representativeness of the testing set-up with respect to the final system implementation. A complete discussion on the concept of testing abstraction and its application to the testing of control-based CPS is found in Section 3 of the paper.

In the paper we identify 4 main classes of testing set-ups: model-, software-, hardware- and process-in-the-loop (see Figure 2 and Table 1 in the paper). We develop a complete drone case study that covers each of them, ensuring the possibility of consistent fault-injection across the set-ups. We then run flight tests for the nominal software and 10 faulty versions of it (the faulty versions are designed to cover different classes of control-software bugs defined in previous literature [Wienke et al., 2016; Steinbauer, 2013]). The flight performances differ across the set-ups (see Table 2 in the paper) and we root-cause the differences for each of them.

From the observations of the flight performances and the root-causing of the differences we draw different conclusions on the relation between the set-ups and their characteristics. Previous literature [Zander et al., 2011], assumed the existence of a hierarchy among the set-ups: i.e. that a set-up with fewer abstracted components can expose strictly more bugs than one with more. Our findings from the test flights confute this assumption (see answer to RQ2 in Section 5 of the paper). Accordingly, we evidence the strengths and weaknesses in fault-finding capabilities of each setup. Here we limit ourselves to reporting that best code coverage is achieved with software-in-the-loop and that timing properties are best tested with hardware-in-the-loop. Conversely, we observed a consistent performance across the set-ups in the verification of functional properties. The complete discussion on said properties can be found in the answer to RQ1 in Section 5 of the Paper. We use these findings to 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 (see answer to RQ2 in Section 5 of the paper).

3.3 Test Case Generation and Oracle (Paper III)

C. Mandrioli, S. Y. Shin, M. Maggio, D. Bianculli, and L. Briand (2022b). “Testing of control-based cyber-physical systems”. *Submitted to ACM Transactions on Software Engineering and Methodology*

Statement of Contribution. The idea of using the limitations of applicability of control-theory to the testing of control-Based CPS was proposed by C. Mandrioli upon discussions with S. Yeob Shin, D. Bianculli, L. Briand, and M. Maggio. C. Mandrioli implemented and executed the case studies. C. Mandrioli took a leading role in writing the paper, while S. Yeob Shin, D. Bianculli, L. Briand, and M. Maggio provided feedback and proofreading.

Scientific Contribution. In Section 1.2 we discussed how the use of feedback both creates intertwining between components of different nature and how it alters the definition of the outputs. In Paper III we address this problem in the context of testing control-based CPS. In control-based CPS, the outputs (i.e., the quantities over which the system requirements can be evaluated) belong to the physical part of the system, instead of being a quantity in the domain of the software (as assumed in traditional software testing). This output alteration, together with the intertwining of components with different nature, makes the development of CPS multidisciplinary. For such systems, we identify three types of engineers involved: the domain engineer, the control-engineer, and the software engineer (as discussed in Section 2.2, Figure 2.2). To address the multidisciplinary nature of the system, we analyse the control design process and discuss in which way the development of the software concerns both control engineers as well as of software engineers. Afterwards, we use this discussion to define the problem of *stress testing control-based CPS software*.

We define stress testing of control-based CPS as the generation of test cases that invalidate to different degrees assumptions made during the design of the system—we list and thoroughly discuss said assumptions in Section 2 of Paper III. In the paper, we identify the different classes of assumptions that control engineers make when they apply control theory. We focus on assumptions related to the linearisation step performed by control engineers (described in Section 2.2). We then use knowledge from control theory to define a qualitative characterization of the input space of a control-based CPS that captures the inputs for which the assumption is expected to be valid. This characterization is described in Section 4 of Paper III: specifically, Figure 5 provides a graphical representation.

The proposed characterization identifies an area of validity where we expect control models to apply. However, the qualitative nature of the characterisation makes it difficult to apply to arbitrary inputs. To overcome this limitation, in the paper we propose a novel test case parametrisation that enables its use. Afterwards, we leverage our characterisation and parametrisation to define metamorphic relations on the tests inputs and outputs, and to develop a testing approach that pushes the CPS to its performance limits.

We empirically evaluate the proposed testing approach on two CPS and inject different sources of non-linearity in one of them: this results in a total of six cases of study. The tests results show that our test case generation approach is able to generate and identify test cases that push the system to its performance limits. Furthermore, we show how the evaluation of the metamorphic relations can be used both as sanity checks for the testing process and also to identify the components that limit the performance CPS.

3.4 Other Publications

The author of this thesis co-authored also the following papers during his PhD studies. They were not included in this thesis to favour it cohesion and flow.

N. Vreman, C. Mandrioli, and A. Cervin (2022). "Deadline-miss-adaptive controller implementation for real-time control systems". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 13–26. DOI: `10.1109/RTAS54340.2022.00010`

P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin (2019). "DMAC: Deadline-Miss-Aware Control". In: S. Quinton (Ed.). *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Vol. 133. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:24. ISBN: 978-3-95977-110-8. DOI: `10.4230/LIPIcs.ECRTS.2019.1`. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/10738`

C. Mandrioli, A. Leva, B. Bernhardsson, and M. Maggio (2019). "Modeling of energy consumption in gps receivers for power aware localization systems". In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '19. Association for Computing Machinery, Montreal, Quebec, Canada, pp. 217–226. ISBN: 9781450362856. DOI: `10.1145/3302509.3311043`. URL: `https://doi.org/10.1145/3302509.3311043`

C. Mandrioli, A. Leva, and M. Maggio (2018). "Dynamic models for the formal verification of big data applications via stochastic model checking". In: *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 1466–1471. DOI: `10.1109/CCTA.2018.8511410`

Bibliography

Abbas, H. and B. Bonakdarpour (2022). “Leveraging system dynamics in runtime verification of cyber-physical systems”.

Abbaspour Asadollah, S., R. Inam, and H. Hansson (2015). “A survey on testing for cyber physical system”. In: El-Fakih, K. et al. (Eds.). *Testing Software and Systems*. Springer International Publishing, Cham, pp. 194–207. ISBN: 978-3-319-25945-1.

Abu-Mostafa, Y. S., M. Magdon-Ismail, and H.-T. Lin (2012). *Learning From Data*. AMLBook. ISBN: 9781600490064.

Aleti, A. and L. Grunske (2015). “Test data generation with a kalman filter-based adaptive genetic algorithm”. *Journal of Systems and Software* **103**, pp. 343–352. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2014.11.035`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121214002660`.

Arcuri, A. and L. Briand (2011). “Adaptive random testing: an illusion of effectiveness?” In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. ACM, Toronto, Ontario, Canada, pp. 265–275. ISBN: 978-1-4503-0562-4. DOI: `10.1145/2001420.2001452`. URL: `http://doi.acm.org/10.1145/2001420.2001452`.

Åstrom, K. J. and R. M. Murray (2008). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA. ISBN: 0691135762.

Ayerdi, J., V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel (2021). “Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Association for Computing Machinery, Athens, Greece, pp. 1264–1274. ISBN: 9781450385626. DOI: `10.1145/3468264.3473920`. URL: `https://doi.org/10.1145/3468264.3473920`.

Balasubramaniam, B., H. Bagheri, S. Elbaum, and J. Bradley (2020). “Investigating controller evolution and divergence through mining and mutation*”. In: *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 151–161. DOI: 10.1109/ICCPS48487.2020.00022.

Banerjee, A., S. Chattopadhyay, and A. Roychoudhury (2016). “Chapter three - on testing embedded software”. In: Memon, A. (Ed.). Vol. 101. Advances in Computers. Elsevier, pp. 121–153. DOI: https://doi.org/10.1016/bs.adcom.2015.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0065245815000662.

Ben Abdessalem, R., S. Nejati, L. C. Briand, and T. Stifter (2018). “Testing vision-based control systems using learnable evolutionary algorithms”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1016–1026. DOI: 10.1145/3180155.3180160.

Böhme, M. (2019). “Assurance in software testing: a roadmap”. In: *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER ’19. IEEE Press, Montreal, Quebec, Canada, pp. 5–8. DOI: 10.1109/ICSE-NIER.2019.00010. URL: https://doi.org/10.1109/ICSE-NIER.2019.00010.

Boyd, J. (1995). *A discourse on winning and losing*. Air University Press Maxwell Air Force Base, AL.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press. DOI: 10.1017/CBO9780511804441.

Bradley, J. M. and H. Bagheri (2020). “Control software: research directions in the intersection of control theory and software engineering”. In: *AIAA Scitech 2020 Forum*. DOI: 10.2514/6.2020-2102. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2020-2102. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2020-2102.

Briand, L., S. Nejati, M. Sabetzadeh, and D. 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, Austin, Texas, pp. 789–792. ISBN: 9781450342056. DOI: 10.1145/2889160.2889212. URL: https://doi.org/10.1145/2889160.2889212.

Bringmann, E. and A. 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, Shanghai, China, pp. 13–20. ISBN: 1595934022. DOI: 10.1145/1138474.1138479. URL: https://doi.org/10.1145/1138474.1138479.

Bringmann, E. and A. Krämer (2008). “Model-based testing of automotive systems”. In: *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 485–493. DOI: `10.1109/ICST.2008.45`.

Broy, M., I. H. Kruger, A. Pretschner, and C. Salzmann (2007). “Engineering automotive software”. *Proceedings of the IEEE* **95**:2, pp. 356–373. DOI: `10.1109/JPROC.2006.888386`.

Bücs, R., L. Murillo, E. Korotcenko, G. Dugge, R. Leupers, G. Ascheid, A. Ropers, M. Wedler, and A. Hoffmann (2016). “Virtual hardware-in-the-loop co-simulation for multi-domain automotive systems via the functional mock-up interface”. In: pp. 3–28. ISBN: 978-3-319-31722-9. DOI: `10.1007/978-3-319-31723-6_1`.

Calafiore, G. C. and M. C. Campi (2006). “The scenario approach to robust control design”. *IEEE Transactions on Automatic Control* **51**:5, pp. 742–753. ISSN: 0018-9286. DOI: `10.1109/TAC.2006.875041`.

Calafiore, G. C. (2013). “Direct data-driven portfolio optimization with guaranteed shortfall probability”. *Automatica* **49**:2, pp. 370–380. ISSN: 0005-1098. DOI: `https://doi.org/10.1016/j.automatica.2012.11.012`. URL: `http://www.sciencedirect.com/science/article/pii/S0005109812005481`.

Cazorla, F. J., T. Vardanega, E. Quiñones, and J. Abella (2013). “Upper-bounding Program Execution Time with Extreme Value Theory”. In: Maiza, C. (Ed.). *13th International Workshop on Worst-Case Execution Time Analysis*. Vol. 30. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 64–76. ISBN: 978-3-939897-54-5. DOI: `10.4230/OASIcs.WCET.2013.64`. URL: `http://drops.dagstuhl.de/opus/volltexte/2013/4123`.

Chen, T. Y., S. C. Cheung, and S. M. Yiu (1998). *Metamorphic testing: a new approach for generating next test cases*. DOI: `10.48550/ARXIV.2002.12543`. URL: `https://arxiv.org/abs/2002.12543`.

Chen, T. Y., F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou (2018). “Metamorphic testing: a review of challenges and opportunities”. *ACM Comput. Surv.* **51**:1. ISSN: 0360-0300. DOI: `10.1145/3143561`. URL: `https://doi.org/10.1145/3143561`.

Chen, T. Y., F.-C. Kuo, R. G. Merkel, and T. H. Tse (2010). “Adaptive random testing: the art of test case diversity”. *J. Syst. Softw.* **83**:1, pp. 60–66. ISSN: 0164-1212. DOI: `10.1016/j.jss.2009.02.022`. URL: `http://dx.doi.org/10.1016/j.jss.2009.02.022`.

Chen, T. Y., F.-C. Kuo, W. K. Tam, and R. G. Merkel (2011). “Testing a software-based pid controller using metamorphic testing”. In: *PECCS*.

Cheng, B. H. C., K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas (2014). “Using models at runtime to address assurance for self-adaptive systems”. In: Bencomo, N. et al. (Eds.). *Models@run.time: Foundations, Applications, and Roadmaps*. Springer International Publishing, Cham, pp. 101–136. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_4. URL: https://doi.org/10.1007/978-3-319-08915-7_4.

Diniz, P. C. and M. C. Rinard (1997). “Dynamic feedback: an effective technique for adaptive computing”. In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. PLDI ’97. Association for Computing Machinery, Las Vegas, Nevada, USA, pp. 71–84. ISBN: 0897919076. DOI: 10.1145/258915.258923. URL: https://doi.org/10.1145/258915.258923.

Dreossi, T., T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh (2015). “Efficient guiding strategies for testing of temporal properties of hybrid systems”. In: Havelund, K. et al. (Eds.). *NASA Formal Methods*. Springer International Publishing, Cham, pp. 127–142. ISBN: 978-3-319-17524-9.

Dutta, S., W. Zhang, Z. Huang, and S. Misailovic (2019). “Storm: program reduction for testing and debugging probabilistic programming systems”. 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, Tallinn, Estonia, pp. 729–739. ISBN: 9781450355728. DOI: 10.1145/3338906.3338972. URL: https://doi.org/10.1145/3338906.3338972.

Filieri, A., C. Ghezzi, A. Leva, and M. Maggio (2011). “Self-adaptive software meets control theory: a preliminary approach supporting reliability requirements”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, Lawrence, KS, USA, pp. 283–292. DOI: 10.1109/ASE.2011.6100064.

Filieri, A., M. Maggio, K. Angelopoulos, N. Dippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel (2017). “Control strategies for self-adaptive software systems”. *ACM Trans. Auton. Adapt. Syst.* **11**:4. ISSN: 1556-4665. DOI: 10.1145/3024188. URL: https://doi.org/10.1145/3024188.

Fisher, R. (1930). *The Genetical Theory of Natural Selection*. OUP Oxford.

Francis, B. and P. Khargonekar (1995). *Robust control theory*. The IMA volumes in mathematics and its applications. Springer-Verlag. ISBN: 9780387944432. URL: https://books.google.se/books?id=81vvAAAAMAAJ.

Fredericks, E. M. and B. H. C. Cheng (2015). “Automated generation of adaptive test plans for self-adaptive systems”. In: *2015 IEEE/ACM 10th International*

Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 157–167. DOI: 10.1109/SEAMS.2015.15.

Gaaloul, K., C. Menghi, S. Nejati, L. C. Briand, and D. Wolfe (2020). “Mining assumptions for software components using machine learning”. 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, Virtual Event, USA, pp. 159–171. ISBN: 9781450370431. DOI: 10.1145/3368089.3409737. URL: https://doi.org/10.1145/3368089.3409737.

Garca, S., D. Strüber, D. Brugali, T. Berger, and P. 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, Virtual Event, USA, pp. 593–604. ISBN: 9781450370431. DOI: 10.1145/3368089.3409743. URL: https://doi.org/10.1145/3368089.3409743.

Garoche, P.-L. (2019). *Formal Verification of Control System Software*. Princeton University Press. ISBN: 9780691181301. URL: http://www.jstor.org/stable/j.ctv80cd4v (visited on 2022-09-14).

Garousi, V., M. Felderer, Ç. M. Karapçak, and U. Ylmaz (2018). “Testing embedded software: a survey of the literature”. *Information and Software Technology* **104**, pp. 14–45. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.06.016. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301265.

Gill, D. H. (2005). “Challenges for critical embedded systems”. In: *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. WORDS ’05. IEEE Computer Society, USA, pp. 7–12. ISBN: 0769523471. DOI: 10.1109/WORDS.2005.21. URL: https://doi.org/10.1109/WORDS.2005.21.

Haan, L. de and A. Ferreira (2010). *Extreme Value Theory: An Introduction (Springer Series in Operations Research and Financial Engineering)*. 1st Edition. Springer. ISBN: 144192020X.

Hänsel, J., T. Vogel, and H. Giese (2015). “A testing scheme for self-adaptive software systems with architectural runtime models”. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pp. 134–139. DOI: 10.1109/SASOW.2015.27.

Hänsel, J., D. Rose, P. Herber, and S. 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*, pp. 170–179. DOI: 10.1109/ICST.2011.37.

Hansen, J., A. Lacis, D. Rind, G. Russell, P. Stone, I. Fung, R. Ruedy, and J. Lerner (1984). "Climate sensitivity: analysis of feedback mechanisms". In: *Climate Processes and Climate Sensitivity*. American Geophysical Union (AGU), pp. 130–163. ISBN: 9781118666036. DOI: `https://doi.org/10.1029/GM029p0130`. eprint: `https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/GM029p0130`. URL: `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/GM029p0130`.

Holler, C., K. Herzig, and A. Zeller (2012). "Fuzzing with code fragments". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. USENIX Association, Bellevue, WA, p. 38.

"ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:General concepts" (2022). *ISO/IEC/IEEE 29119-1:2022(E)*, pp. 1–60. DOI: `10.1109/IEEESTD.2022.9698145`.

J N Reddy, D. (2005). *An Introduction to the Finite Element Method*. McGraw-Hill Education. ISBN: 9780072466850. URL: `https://books.google.se/books?id=8ofqngEACAAJ`.

Jiménez-Martín, A., A. Mateos, and S. Ríos-Insua (2005). "Monte carlo simulation techniques in a decision support system for group decision making". *Group Decision and Negotiation* **14**, pp. 109–130. DOI: `10.1007/s10726-005-2406-9`.

Johnson, O. (2004). *Information Theory and the Central Limit Theorem*. Imperial College Press. ISBN: 9781860945373. URL: `https://books.google.se/books?id=r5XI8a0lYykC`.

Kim, J., H. Kim, K. Lakshmanan, and R. (Rajkumar (2013). "Parallel scheduling for cyber-physical systems: analysis and case study on a self-driving car". In: *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ICCPS '13. Association for Computing Machinery, Philadelphia, Pennsylvania, pp. 31–40. ISBN: 9781450319966. DOI: `10.1145/2502524.2502530`. URL: `https://doi.org/10.1145/2502524.2502530`.

Lamberg, K., M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey (2004). "Model-based testing of embedded automotive software using mtest". In: *SAE 2004 World Congress and Exhibition*. SAE International. DOI: `https://doi.org/10.4271/2004-01-1593`. URL: `https://doi.org/10.4271/2004-01-1593`.

Lan, H. and Y. Sun (2021). "Review on fuzz testing for protocols in industrial control systems". In: *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, pp. 433–438. DOI: `10.1109/DSC53577.2021.00068`.

Lee, E. A. (2015). "The past, present and future of cyber-physical systems: a focus on models". *Sensors* **15**:3, pp. 4837–4869. ISSN: 1424-8220. DOI: `10.3390/s150304837`. URL: `https://www.mdpi.com/1424-8220/15/3/4837`.

Mandrioli, C., A. Leva, B. Bernhardsson, and M. Maggio (2019). “Modeling of energy consumption in gps receivers for power aware localization systems”. In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS ’19. Association for Computing Machinery, Montreal, Quebec, Canada, pp. 217–226. ISBN: 9781450362856. DOI: `10.1145/3302509.3311043`. URL: `https://doi.org/10.1145/3302509.3311043`.

Mandrioli, C., A. Leva, and M. Maggio (2018). “Dynamic models for the formal verification of big data applications via stochastic model checking”. In: *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 1466–1471. DOI: `10.1109/CCTA.2018.8511410`.

Mandrioli, C. and M. Maggio (2020). “Testing self-adaptive software with probabilistic guarantees on performance metrics”. In: *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM. ISBN: 9781450370431. DOI: `10.1145/3368089.3409685`.

Mandrioli, C. and M. Maggio (2022). “Testing self-adaptive software with probabilistic guarantees on performance metrics: extended and comparative results”. *IEEE Transactions on Software Engineering* **48**:9, pp. 3554–3572. DOI: `10.1109/TSE.2021.3101130`.

Mandrioli, C., M. Nyberg Carlsson, and M. Maggio (2022a). “Testing abstractions for cyber-physical control systems”. *Submitted to ACM Transactions on Software Engineering and Methodology*.

Mandrioli, C., S. Y. Shin, M. Maggio, D. Bianculli, and L. Briand (2022b). “Testing of control-based cyber-physical systems”. *Submitted to ACM Transactions on Software Engineering and Methodology*.

Marculescu, B., R. Feldt, R. Torkar, and S. Poulding (2015). “An initial industrial evaluation of interactive search-based testing for embedded software”. *Applied Soft Computing* **29** (0), pp. 26–39. DOI: `http://dx.doi.org/10.1016/j.asoc.2014.12.025`. URL: `http://www.sciencedirect.com/science/article/pii/S1568494614006693`.

Matinnejad, R., S. Nejati, L. Briand, and T. 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, Vasteras, Sweden, pp. 163–174. ISBN: 9781450330138. DOI: `10.1145/2642937.2642978`. URL: `https://doi.org/10.1145/2642937.2642978`.

Matinnejad, R., S. Nejati, and L. 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, Paderborn, Germany, pp. 938–

943. ISBN: 9781450351058. DOI: 10.1145/3106237.3117770. URL: https://doi.org/10.1145/3106237.3117770.

Maxwell, J. C. (2011). “On governors”. In: Niven, W. D. (Ed.). *The Scientific Papers of James Clerk Maxwell.* Vol. 2. Cambridge Library Collection - Physical Sciences. Cambridge University Press, pp. 105–120. DOI: 10.1017/CBO9780511710377.009.

Mehmood, M. A., M. N. A. Khan, and W. Afzal (2018). “Automating test data generation for testing context-aware applications”. In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 104–108. DOI: 10.1109/ICSESS.2018.8663920.

Menghi, C., S. Nejati, L. C. Briand, and Y. I. Parache (2019). “Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification”. *CoRR* **abs/1910.02837**. arXiv: 1910.02837. URL: http://arxiv.org/abs/1910.02837.

Message from the 50 Years of Software Engineering Chairs (2018), pp. 30–30.

Micskei, Z., Z. Szatmári, J. Oláh, and I. Majzik (2012). “A concept for testing robustness and safety of the context-aware behaviour of autonomous systems”. In: *Proceedings of the 6th KES International Conference on Agent and Multi-Agent Systems: Technologies and Applications*. KES-AMSTA12. Springer-Verlag, Dubrovnik, Croatia, pp. 504–513. ISBN: 9783642309465. DOI: 10.1007/978-3-642-30947-2_55. URL: https://doi.org/10.1007/978-3-642-30947-2_55.

Myers, G. (1979). *The Art of Software Testing*. Business Data Processing: A Wiley Series. Wiley. ISBN: 9780471043287. URL: https://books.google.se/books?id=DV0ZAQAAIAAJ.

Nejati, S., K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. 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, Tallinn, Estonia, pp. 1015–1025. ISBN: 9781450355728. DOI: 10.1145/3338906.3340444. URL: https://doi.org/10.1145/3338906.3340444.

Nielsen, C. B., P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska (2015). “Systems of systems engineering: basic concepts, model-based techniques, and research directions”. *ACM Comput. Surv.* **48**:2. ISSN: 0360-0300. DOI: 10.1145/2794381. URL: https://doi.org/10.1145/2794381.

Oliveira Neves, V. de, A. Bertolino, G. De Angelis, and L. Garcés (2018). “Do we need new strategies for testing systems-of-systems?” In: *Proceedings of the 6th International Workshop on Software Engineering for Systems-of-Systems*. SESoS ’18. ACM, Gothenburg, Sweden, pp. 29–32. ISBN: 978-1-4503-5747-0.

DOI: 10.1145/3194754.3194758. URL: http://doi.acm.org/10.1145/3194754.3194758.

Pazzaglia, P., C. Mandrioli, M. Maggio, and A. Cervin (2019). "DMAC: Deadline-Miss-Aware Control". In: Quinton, S. (Ed.). *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Vol. 133. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:24. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.1. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10738.

Peleska, J. (2002). "Hardware/software integration testing for the new airbus aircraft families." *http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html*. DOI: 10.1007/978-0-387-35497-2_24.

Plaku, E., L. E. Kavraki, and M. Y. Vardi (2009). "Falsification of ltl safety properties in hybrid systems". In: Kowalewski, S. et al. (Eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 368–382. ISBN: 978-3-642-00768-2.

Qin, Y., C. Xu, P. Yu, and J. Lu (2016). "Sit: sampling-based interactive testing for self-adaptive apps". *Journal of Systems and Software* **120**, pp. 70–88. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.07.002. URL: http://www.sciencedirect.com/science/article/pii/S0164121216301029.

Ramponi, F. A. and M. C. Campi (2018). "Expected shortfall: heuristics and certificates". *European Journal of Operational Research* **267**:3, pp. 1003–1013. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2017.11.022. URL: http://www.sciencedirect.com/science/article/pii/S0377221717310330.

Rawung, R. H. and A. G. Putrada (2014). "Cyber physical system: paper survey". In: *2014 International Conference on ICT For Smart Society (ICISS)*, pp. 273–278. DOI: 10.1109/ICTSS.2014.7013187.

Reichstaller, A. and A. Knapp (2018). "Risk-based testing of self-adaptive systems using run-time predictions". In: *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 80–89. DOI: 10.1109/SASO.2018.00019.

Robert, C. P. and G. Casella (2005). *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag, Berlin, Heidelberg. ISBN: 0387212396.

Robert, C. P. and G. Casella (2010). "Monte carlo optimization". In: *Introducing Monte Carlo Methods with R*. Springer New York, New York, NY, pp. 125–165. ISBN: 978-1-4419-1576-4. DOI: 10.1007/978-1-4419-1576-4_5. URL: https://doi.org/10.1007/978-1-4419-1576-4_5.

Salama, M. and R. Bahsoon (2017). "Analysing and modelling runtime architectural stability for self-adaptive software". *Journal of Systems and Software* **133**, pp. 95–112. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2017.07.041`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121217301620`.

Salehie, M. and L. Tahvildari (2009). "Self-adaptive software: landscape and research challenges". *ACM Trans. Auton. Adapt. Syst.* **4**:2. ISSN: 1556-4665. DOI: `10.1145/1516533.1516538`. URL: `https://doi.org/10.1145/1516533.1516538`.

Santinelli, L., J. Morio, G. Dufour, and D. Jacquemart (2014). "On the sustainability of the extreme value theory for wcet estimation". In: *OpenAccess Series in Informatics*. Vol. 39. DOI: `10.4230/OASIcs.WCET.2014.21`.

Santos, I. d. S. (2017). *TESTDAS: Testing MEthod for Dynamically Adaptive Systems*. PhD thesis. Universisdade Federal do Ceara, Fortaleza, Brazil.

Shen, Q., M. Wen, L. Zhang, L. Wang, L. Shen, and J. Cheng (2021). "A systematic review of fuzzy testing for information systems and applications". In: *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, pp. 156–162. DOI: `10.1109/CECIT53797.2021.00035`.

Silano, G., E. Aucone, and L. 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)*, pp. 1–6. DOI: `10.1109/MED.2018.8442759`.

Silva, C. E. da and R. de Lemos (2011). "Dynamic plans for integration testing of self-adaptive software systems". In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. Association for Computing Machinery, Waikiki, Honolulu, HI, USA, pp. 148–157. ISBN: 9781450305754. DOI: `10.1145/1988008.1988029`. URL: `https://doi.org/10.1145/1988008.1988029`.

Siqueira, B. R., F. C. Ferrari, M. A. Serikawa, R. Menotti, and V. V. de Camargo (2016). "Characterisation of challenges for testing of adaptive systems". In: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*. SAST. Association for Computing Machinery, Maringa, Parana, Brazil. ISBN: 9781450347662. DOI: `10.1145/2993288.2993294`. URL: `https://doi.org/10.1145/2993288.2993294`.

Steinbauer, G. (2013). "A survey about faults of robots used in robocup". In: Chen, X. et al. (Eds.). *RoboCup 2012: Robot Soccer World Cup XVI*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 344–355. ISBN: 978-3-642-39250-4.

Timperley, C. S., 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)*, pp. 331–342. DOI: `10.1109/ICST.2018.00040`.

Tramontana, P., D. Amalfitano, N. Amatucci, A. Memon, and A. R. Fasolino (2019). "Developing and evaluating objective termination criteria for random testing". *ACM Trans. Softw. Eng. Methodol.* **28**:3, 17:1–17:52. ISSN: 1049-331X. DOI: 10.1145/3339836. URL: http://doi.acm.org/10.1145/3339836.

Vreman, N., C. Mandrioli, and A. Cervin (2022). "Deadline-miss-adaptive controller implementation for real-time control systems". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 13–26. DOI: 10.1109/RTAS54340.2022.00010.

Wang, H., W. K. Chan, and T. H. Tse (2014). "Improving the effectiveness of testing pervasive software via context diversity". *ACM Trans. Auton. Adapt. Syst.* **9**:2. ISSN: 1556-4665. DOI: 10.1145/2620000. URL: https://doi.org/10.1145/2620000.

Weyns, D. (2017). "Software engineering of self-adaptive systems: an organised tour and future challenges". *Chapter in Handbook of Software Engineering*, p. 2.

White, A. (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. DOI: 10.1109/AERO.2001.931302.

Wienke, J., S. Meyer zu Borgsen, and S. Wrede (2016). "A data set for fault detection research on component-based robotic systems". In: Alboul, L. et al. (Eds.). *Towards Autonomous Robotic Systems*. Springer International Publishing, Cham, pp. 339–350. ISBN: 978-3-319-40379-3.

Wong, T., M. Wagner, and C. Treude (2022). "Self-adaptive systems: a systematic literature review across categories and domains". *Information and Software Technology* **148**, p. 106934. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2022.106934. URL: https://www.sciencedirect.com/science/article/pii/S0950584922000854.

Wu, M., H. Zeng, C. Wang, and H. Yu (2017). "Invited: safety guard: runtime enforcement for safety-critical cyber-physical systems". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6. DOI: 10.1145/3061639.3072957.

Yamagata, Y., S. Liu, T. Akazaki, Y. Duan, and J. Hao (2021). "Falsification of cyber-physical systems using deep reinforcement learning". *IEEE Transactions on Software Engineering* **47**:12, pp. 2823–2840. DOI: 10.1109/TSE.2020.2969178.

Yatoh, K., K. Sakamoto, F. Ishikawa, and S. Honiden (2015). "Feedback-controlled random test generation". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Association for Computing Machinery, Baltimore MD USA, pp. 316–326. ISBN: 9781450336208. DOI: 10.1145/2771783.2771805. URL: https://doi.org/10.1145/2771783.2771805.

Yu, L., W. T. Tsai, Y. Jiang, and J. Gao (2014). “Generating test cases for context-aware applications using bigraphs”. In: *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pp. 137–146. DOI: `10.1109/SERE.2014.27`.

Yun, J., F. Rustamov, J. Kim, and Y. Shin (2022). “Fuzzing of embedded systems: a survey”. *ACM Comput. Surv.* Just Accepted. ISSN: 0360-0300. DOI: `10.1145/3538644`. URL: `https://doi.org/10.1145/3538644`.

Zander, J., I. Schieferdecker, and P. Mosterman (2011). *Model-Based Testing for Embedded Systems*. ISBN: 9781439818459.

Zeller, A., R. Gopinath, M. Böhme, G. Fraser, and C. Holler (2019). “The fuzzing book”. In: *The Fuzzing Book*. Retrieved 2019-09-09 16:42:54+02:00. Saarland University. URL: `https://www.fuzzingbook.org/`.

Zhou, X., X. Gou, T. Huang, and S. Yang (2018). “Review on testing of cyber physical systems: methods and testbeds”. *IEEE Access* **6**, pp. 52179–52194. DOI: `10.1109/ACCESS.2018.2869834`.

Zimmer, M., J. Hedrick, and E. A. Lee (2015). “Ramifications of software implementation and deployment: a case study on yaw moment controller design”. *2015 American Control Conference (ACC)*, pp. 2014–2019.

Paper I

Testing Self-Adaptive Software with Probabilistic Guarantees on Performance Metrics: Extended and Comparative Results

Claudio Mandrioli, Martina Maggio

Abstract

This paper discusses methods to test the performance of the adaptation layer in a self-adaptive system. The problem is notoriously hard, due to the high degree of uncertainty and variability inherent in an adaptive software application. In particular, providing any type of formal guarantee for this problem is extremely difficult. In this paper we propose the use of a rigorous probabilistic approach to overcome the mentioned difficulties and provide probabilistic guarantees on the software performance. We describe the set up needed for the application of a probabilistic approach. We then discuss the traditional tools from statistics that could be applied to analyse the results, highlighting their limitations and motivating why they are unsuitable for the given problem. We propose the use of a novel tool – *the Scenario Theory* – to overcome said limitations. We conclude the paper with a thorough empirical evaluation of the proposed approach, using three adaptive software applications: the Tele-Assistance Service, the Self-Adaptive Video Encoder, and the Traffic Reconfiguration via Adaptive Participatory Planning. With the first, we empirically expose the trade-off between data collection and confidence in the testing campaign. With the second, we demonstrate how to compare different adaptation strategies. With the third, we discuss the role of the randomisation in the selection of test inputs. In the evaluation, we apply the scenario theory and also classical statistical tools: Monte Carlo and Extreme Value Theory. We provide a complete evaluation and a thorough comparison of the confidence and guarantees that can be given with all the approaches.

Originally published in Transactions of Software Engineering. Reprinted with permission.

1. Introduction

Software systems are affected by uncertainty that alters their behaviour and can render their performance unpredictable. Adaptation layers were introduced in software as a viable solution to deal with performance fluctuations and minimise the effect of uncontrolled changes [Salehie and Tahvildari, 2009; Cheng et al., 2009; Cheng et al., 2014]. This makes software self-adaptive. The idea behind self-adaptive software is to have a layer responsible for observing behavioural changes and taking counteractions. This can guarantee more stable and predictable software performance in terms of non-functional software behaviour [Filieri et al., 2011; Moreno et al., 2015], e.g., lower response times, or higher reliability.

Adaptation can be implemented using different methodologies; some of them provide guarantees based on formal models [Filieri et al., 2014; Moreno et al., 2015; Gulisano et al., 2017], others are empirically proven effective [DIppolito et al., 2014; Shevtsov and Weyns, 2016; Moreno et al., 2017]. In both cases there is a need for appropriate performance testing of the system composed of the software and its adaptation layer. The presence of an adaptation layer opens up the possibility that in the same exact condition the software will behave differently, depending on its past behaviour and accumulated knowledge. It is necessary to conduct empirical validation of satisfactory behaviour to verify the correctness of the system and adaptation-layer implementation [Weyns, 2012]. In addition, it is important to quantify the achievable performance.

In general, testing is a crucial aspect of software development. For self-adaptive software, the testing process is complicated by the presence of the adaptation layer [Briand et al., 2016; González et al., 2018; Siqueira et al., 2016; Bertolino and Inverardi, 2019]. Self-adaptive systems testing is intrinsically hard, due to the extreme variability and uncertainty involved in the software execution [Siqueira et al., 2016; Oliveira Neves et al., 2018; Bahar et al., 2019; Bertolino et al., 2003]. In fact, the adaptation layer explicitly reacts to the uncertainty, and may influence it for the future. This creates a *loop* around the software [Salehie and Tahvildari, 2009]. In the context of uncertainty and adaptation, this paper's challenge is to achieve and maintain formal guarantees on non-functional aspects of the software execution, such as reliability and response times.

Research Challenges: The adaptation layer and the presence of uncertainty impose specific challenges for testing. Triggered by the environmental variability, the adaptation generates changes in the system - and this changing nature makes it difficult (and in many cases impossible) to *exhaustively* guarantee its correct behaviour [Siqueira et al., 2016; Munoz and Baudry, 2009; Tse et al., 2004; Welsh and Sawyer, 2010; Micskei et al., 2012; Ferrari et al., 2011]. The adaptation also creates a difficulty in the performance quantification and in determining the testing sufficiency and effectiveness [Oliveira Neves et al., 2018; Siqueira et al., 2016].

As an example, consider testing a web-application that can run on different servers with different and time-varying performance results. Not all of the servers

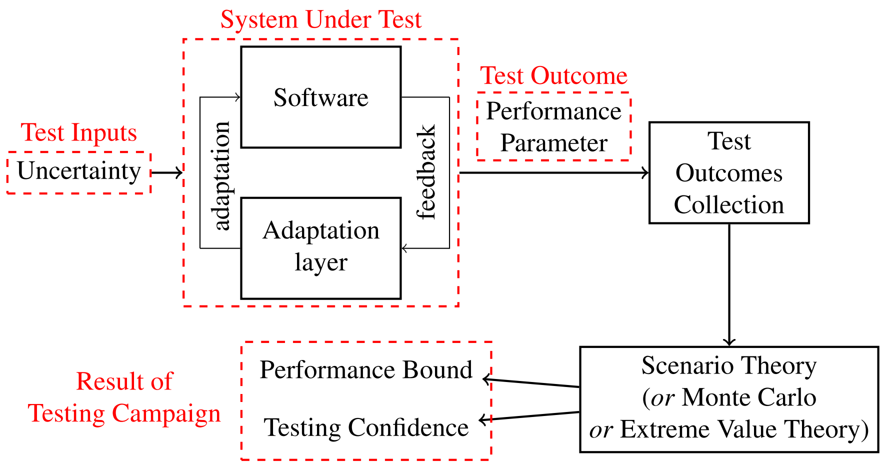


Figure 1. Overview of the proposed approach. In the figure, black blocks represent components of the adaptive system and of the testing procedure. The red text and boxes highlight the main high-level concepts of the testing procedure. The arrows represent information flow. Figure from [Mandrioli and Maggio, 2020].

can provide the same reliability. The adaptive layer should choose dynamically which server to use, in order to maximise the overall reliability. In general, it is not possible to guarantee that the application is always reliable, since any server may fail. Also, the actual reliability will depend significantly on the specific servers, and on their performance. As a consequence, when testing the system, any evaluation of its reliability is heavily affected by the specific test cases. Determining which tests are sufficient and when it is possible to stop the testing process becomes challenging. Prior literature contributions highlighted a set of research challenges for testing adaptive software [Siqueira et al., 2016; Oliveira Neves et al., 2018]. The quoted papers listed a series of challenges, and then group them into macro areas: types of guarantees, quantification of performance metrics, and quantification of the testing effectiveness. In this paper, we try to address these macro challenges:

- **CH1**: Definition of what type of guarantees can be given for self-adaptive software.
- **CH2**: Quantification of the mentioned guarantees.
- **CH3**: Quantification of the testing sufficiency and effectiveness (or testing adequacy).

Furthermore, any method based on statistics heavily depends on the input data. In this paper we also discuss an additional challenge (that is mentioned in [Siqueira et al., 2016; Oliveira Neves et al., 2018] with less emphasis):

- **CH4**: Definition and collection of testing input data.

We believe **CH1** to be the main challenge that we try to solve in this paper. The remaining challenges can be viewed as sub-challenges, that mark relevant aspects in the definition of a methodology to address CH1. Towards a solution for **CH1**, we define a particular notion of probabilistic guarantees that can be provided after a testing phase of self-adaptive strategies and software. The need to provide such formal guarantees, forces us to think and consider the other relevant aspects. Probabilistic guarantees trigger the need for repeated testing, resulting in the need of generating randomised test cases and to carefully select input data, therefore triggering **CH4**. When tests are performed, the data gathered in the testing phase should be analysed to quantify the provided guarantees, giving an answer to **CH2**. It is also important to quantify the testing adequacy – as mentioned in **CH3** – i.e. to understand if the number of conducted tests is sufficient or if more tests are needed. This clearly depends on the nature and type of the probabilistic guarantees to be produced.

Contribution: In this paper we address the four mentioned research challenges by leveraging a rigorous probabilistic approach [Böhme, 2019; Dutta et al., 2019]. The probabilistic approach is beneficial in two ways: (i) it allows the efficient exploration of large input and configuration spaces [Robert and Casella, 2005], and (ii) it can provide a quantification of its own adequacy. In the field of probability theory, the testing adequacy is called *confidence*.

As discussed in the research challenges above, the uncertain nature of self-adaptive systems does not allow for the definition of strict guarantees. This limitation mainly arises from the large (and possibly infinite) number of combinations of inputs that can be provided to the system [Siqueira et al., 2016; Oliveira Neves et al., 2018]. Despite this, *we need to test self-adaptive systems when said variability is present, in order to trigger the adaptive behaviour*. Leveraging a probabilistic approach, the uncertainty and variability can efficiently be explored using randomised inputs. As a consequence, the measured performance metric must be treated as a random quantity, and requires statistical evaluation. We therefore enter the domain of probabilistic guarantees [Böhme, 2019; Abu-Mostafa et al., 2012]. The randomised approach allows for an efficient exploration that is independent from the size and quantity of the uncertainty that is present in the software execution [Robert and Casella, 2005; Dutta et al., 2019].

In this work, we focus on the evaluation of probabilistic bounds for a given performance metric. In contrast with *conventional* testing, *statistical* testing can only provide confidence values. In conventional testing, a property is either evaluated to true or false. When the same test is repeated with a different set of random input values, the property evaluation becomes a stochastic variable. This means that the result of the testing process is the confidence in the property being either true or false, when repeating the test with a new input set.

According to the probabilistic framework, our aim is testing what is the value of the performance parameter that the adaptive software can guarantee in the "majority" of its execution environments. We formally define majority in a probabilistic fashion, e.g., that a given performance bound will hold in 99% of the execution instances. We also quantify the confidence that we can claim, i.e., the adequacy of our testing campaign. High confidence means a high probability that we performed a sufficient number of randomly generated tests to sustain our claim. In some sense, this is analogous to a *coverage criterion* – a reference for choosing when to stop the testing campaign. One of the contributions of the paper is the discussion of input data. We provide guidelines on what needs to be randomised and what – in contrasts – remains fixed over the set of executed tests. We also discuss testing a partial system in contrast to testing the real system in operation, where there is no need for randomisation of input data because the real execution uses a *realistic* set of execution conditions.

In the paper, we discuss traditional tools from statistics (Monte Carlo and Extreme Value Theory) and highlight their limitations for testing self-adaptive software. We overcome these limitations using a tool called *Scenario Theory* [Calafiore and Campi, 2006]. The Scenario Theory was developed in the field of robust control but can actually be applied to a very general class of problems. In this paper we show how to apply it to the problem of testing self-adaptive software. We provide a thorough comparison between the confidence and the results obtained with the Scenario Theory and with Monte Carlo and Extreme Value Theory.

Experimental Evaluation: To support our claims, we use our methodology to test the behaviour of three self-adaptive software applications: the Tele-Assistance System [Weyns and Calinescu, 2015], the Self-Adaptive Video Encoder [Maggio et al., 2017b], and the Traffic Reconfiguration via Adaptive Participatory Planning [Gerostathopoulos and Pournaras, 2019]. We show the complete application of Monte Carlo, Extreme Value Theory and the Scenario Theory. In all cases, we discuss how these methods can be used to: (i) rigorously quantify the adaptation performance, (ii) evaluate the trade-off between the number of performed tests and the confidence in the testing campaign, and (iii) compare adaptation strategies.[1] Our experimental results show that the Scenario Theory provides better guarantees and higher confidence in the results.

Extension: This paper extends our previous work [Mandrioli and Maggio, 2020] providing novel contribution, both on the methodological discussion, and on the empirical evaluation. For what concerns the methodology, we present a detailed discussion of the application of traditional statistical tools to the testing of self-adaptive systems. We also apply the mentioned tools in our empirical evaluation, comparing the results obtained with our proposed testing strategy. We included an additional case study to evaluate the methodology on a different software application, in particular with respect to the relevance of the randomisation of the input

[1] The implementation of the experiments presented in the paper is publicly available and has been reproduced through the conference artifact review process `https://github.com/ManCla/ESEC-FSE-2020` [Martina Maggio, 2020].

data.

Paper Structure: In Section 2 we provide an overview of our proposed testing approach. Section 3 discusses related work. Section 4 presents our methodology and describes how it overcomes the limitations of classical statistical testing. Section 5 presents experimental results. Finally, Section 6 discusses the limitations and threats to validity of the proposed approach and Section 7 concludes the paper.

2. Approach Overview

In this section we provide an overview of our testing approach (shown in Figure 1). In particular, we discuss: (i) the definition of *test inputs*, (ii) the definition of the *test outcome*, and (iii) the evaluation of the *results of the testing campaign*. In Sections 3.2 and 4 we discuss in detail how to apply respectively traditional tools and the Scenario Theory to evaluate the test outcomes and obtain *performance bounds* and *testing confidence*.

The objective of our testing campaign is to empirically provide guarantees on the system behaviour. These guarantees should be general and independent of the specific test cases. Practically, we want independence from the variability and uncertainty that affects the executed tests. We obtain this by performing different random tests, each of which represents a possible system realisation. We then statistically evaluate the results of the testing campaign.

Performing repeated random tests requires, as a first step, the definition of what are the test inputs that need randomisation. The remaining inputs, instead, should be fixed across the tests. Using the web application example, we can say that the number of connected users is a parameter that should vary from one test to the next, but (possibly) the amount of threads that are assigned to serving requests from clients is fixed and defined by the specification of the software architecture.

The choice of what to randomise and what to keep constant highly impacts the significance of the testing campaign. If more than necessary test inputs are randomised, the testing results can be unnecessarily conservative. Conversely, fixing inputs that will actually vary in the actual software execution – and therefore are uncertain and unknown – will provide results that do not carry on between the testing campaign and the actual software implementation.

The testing engineer should randomise all inputs that are not known at development time and that will affect the system performance. Most importantly, these have to include the exogenous inputs that the software is supposed to adapt to. In this way, the random sampling will trigger the adaptation layer and explore the possible performances of the system. From a practical point of view, the randomised inputs should be all of those inputs that will make the same implementation of the system potentially behave at a different performance level. The definition and analysis of the relevant test inputs are domain and application dependant: different types of adaptive software are required to respond to different inputs (e.g. discrete signals or continuous quantities). Moreover, an evaluation on the quality of the test inputs requires assumptions and modeling efforts so that an analysis can be carried on. Apparently, this implies that a significant effort is required to the testing engineer for the definition of the testing inputs. On the contrary, one of the main strengths of ST is to include a coverage criteria that requires minimal assumptions on the testing inputs definition without loss of guarantees.

Exhaustively listing all the system inputs in an adaptive system can be a difficult task [Trubiani and Apel, 2019]. In some cases, the test input definition problem can be circumvented by executing the actual system with the actual inputs, rather using synthetic random inputs. In the web application example, one could collect traces from the execution of the actual software and analyse this data as a set of random test cases. If the executions are collected systematically (i.e., two different system executions will differ exclusively by the unknown inputs), the uncertain inputs are in this way *by definition* representative of a possible realization of the adaptive system. This is a viable solution thanks to the minimal assumptions on the distribution of the inputs required by ST.For this reason, in this paper we build on an approach that does not require any of such assumptions. As we discuss formally in Section 4, the fact of not requiring any assumption on the input probability distributions is one of the strengths of the proposed scenario theory.

Conversely, in our testing set-up, all the inputs that are known and fixed once the system is implemented should not be randomised. In the web application example, this could be the number of servers on which the applications is deployed. If the number is known and fixed at deployment-time, their number should be fixed also in the tests. Otherwise, if the application is expected to adapt to a varying number of severs, their number should be randomised during and across the tests.

In order to evaluate the effectiveness of the adaptation strategy, we define a *performance parameter* that we compute for each test. The performance parameter is a quantity that (i) can be measured from the execution of a test case, and (ii) is higher or lower, according to the degree at which the adaptation strategy has achieved its goals. In the web-application case mentioned above, this parameter could be, for example, the average time spent recovering from server failures over the whole test duration. The key intuition is that this performance parameter is itself a random variable [Böhme, 2019], and we can therefore use tools from statistics to deduce properties of its value.

The extraction of such properties can be done in different ways. Traditional statistics offers different tools that could be considered. We argue that these tools present fundamental limitations that hinder their applicability to test self-adaptive software. As an alternative, we propose the use of scenario theory.

Hence, we collect the outcomes of the tests and evaluate them using the scenario theory. By leveraging this theory we obtain *probabilistic bounds* on the chosen performance metric and a *testing confidence*. The probabilistic bounds are in the form of a minimum performance that is guaranteed in a high percentage of the cases. The testing confidence is given as a probability. To be precise, the confidence is the probability that we have missed relevant test cases that would have changed the obtained bound. Continuing with the example above, we would obtain a bound like "*the time it takes to recover from a server failure is on average less than* 42 *seconds in* 97% *of the cases, with a* 95% *confidence*". This means that we have a $100 - 95 = 5\%$ probability of having missed a relevant test case. If the confidence is not sufficient, the scenario theory allows the testing engineer to directly compute how many additional tests are needed to increase it to the desired level.

3. Background and Related work

This section discusses how this work is connected to the existing research literature. To start, we present related work in the software testing research field. Then we present the traditional statistical tools used to extract probabilistic properties from test outcomes.

3.1 Testing of Adaptive Systems

Our work connects to different areas of the existing software testing literature: (i) testing of self-adaptive and context-aware systems, (ii) testing in the presence of environmental dependencies, (iii) fuzz testing, and (iv) testing for probabilistic guarantees.

The problem of testing an adaptive software – in some cases also called *context-aware* software [Wang et al., 2014; Micskei et al., 2012] – is not a new challenge for the software testing community [Siqueira et al., 2016; Oliveira Neves et al., 2018]. We split the work that addresses the testing of self adaptive software in *design-time* and *run-time* approaches. For self-adaptive software, the design-time approaches include SIT [Qin et al., 2016] and TestDAS [Santos, 2017]. SIT [Qin et al., 2016] proposes a test case generation technique for self-adaptive applications. The sampling of the input space is based on an interactive model of the application that is being tested. TestDAS [Santos, 2017] focuses on triggering the adaptations during the test cases. It leverages models of the software behaviour that are defined in advance by the programmer. Context-aware software is close to self-adaptive software, and there is a significant amount of work addressing the problem of testing context-aware applications [Wang et al., 2014; Yu et al., 2014; Mehmood et al., 2018; Micskei et al., 2012]. The self-adaptive (or context-aware) software observes the execution environment and selects actions to be performed based on the result of the observation phase. The research effort for context-aware software goes in the direction of generating test cases that trigger the context-aware software layer [Yu et al., 2014; Mehmood et al., 2018; Micskei et al., 2012]. In [Yu et al., 2014], automatically generated bigraphs are used to model the interactions between the environment and the software, and to generate the test cases. In [Mehmood et al., 2018] the authors propose a framework for automatically generating test cases with high-level test data.

Our proposal is different from previous work on context-aware and self-adaptive software testing, since in our case the interaction with the environment only needs a probabilistic characterisation, and no further modelling effort. Moreover, in our contribution, the number of test cases does not depend on how the interaction with the environment is performed. This is important since it allows our method to scale with the amount of interaction between software and environment.

The literature on software testing also includes efforts to develop run-time testing methodologies for adaptive software [Cheng et al., 2014; Hänsel et al., 2015; Reichstaller and Knapp, 2018]. Generally speaking, there is a need to develop models for verification and validation at run-time [Cheng et al., 2014]. This need is caused by the ever-changing nature of the environment the adaptive software operates in. We describe our approach for design-time testing, but in principle[2] the resulting method can be applied during the run-time execution of the software application, since it only requires data collection and analysis. A clear difference between our work and the related literature is that we develop a probabilistic approach.

In our work, we use statistical tools to evaluate the performance of the adaptation layer of a self-adaptive software, independently from changes in the environment. Previous work also addressed the problem of testing a software regardless of its environmental dependencies [Arcuri et al., 2014; Hervieu et al., 2012]. These works aim at decoupling the tests outcomes from such dependencies. To test the adaptation layer, we need to preserve the dependency on the environment, since it triggers the need for adaptation. However, we aim at obtaining an evaluation that is general with respect to the environment changes.

The approach we propose in this paper is based on random sampling of the system inputs and environment scenarios. This practice is known to the software testing community [Chen et al., 2010; Arcuri and Briand, 2011], and is often called fuzz testing [Zeller et al., 2019; Yatoh et al., 2015; Tramontana et al., 2019; Böhme, 2019]. The literature focuses on using random generation for achieving adequate exploration of the software behaviour, e.g., code coverage [Tramontana et al., 2019]. We take inspiration from fuzz testing, and use random sampling with two different objectives: (i) decoupling given inputs or environmental scenarios from performance parameters that indicate how well the adaptation layer is performing, and (ii) obtaining a probabilistic characterisation of the performance metric.

Probabilistic guarantees have been explored [Hierons and Merayo, 2009; Rosario et al., 2008; Hwang et al., 2007]. In some cases this exploration targeted approximate computing [Dutta et al., 2019; Bahar et al., 2019; Joshi et al., 2019; Dutta et al., 2018], which is not the subject of this study. Some existing work target service-oriented software architectures [Canfora and Di Penta, 2006] and how to combine the probabilistic guarantees given by the different services to obtain guar-

[2] The requirement to apply our approach at run-time is that the run-time tests are considered random independent tests. Testing the system continuously might not guarantee independence. This can be solved (for example) by introducing a delay between consecutive tests.

antees for the complete system [Rosario et al., 2008; Hwang et al., 2007]. Recent work used a probabilistic approach to compensate for the uncertainty of the dependence between system configurations and system performance [Dorn et al., 2020]. However, no prior work targets dynamic behaviour (i.e., behaviour that changes during the execution of the software, as it is the case with the adaptation layer) and adaptive software, which is the focus of our work.

3.2 Tools from Statistics

In this section, we recall traditional tools from statistics, that could be used to analyse the result of tests and provide statistical guarantees on the software behaviour. We both describe the underlying theory and their application. A throughout discussion of the limitations that make them unsuitable for the testing of self-adaptive software is later presented in Section 4.1. We choose Monte Carlo Sampling (MC) and Extreme Value Theory (EVT) due to their application in software testing, simulations, and rare event analysis. Statistics offer additional tools, e.g., martingale and black swan theory, that are less suited to the analysis of a vast corpus of data or require additional knowledge. We have not found evidence of application of other theoretical results to the field of software testing.

Monte Carlo Sampling (MC): Monte Carlo (MC) methods [Robert and Casella, 2005] use repeated random sampling and simulation to numerically predict the value of parameters. The parameters are unknown, and usually no exact analysis can be carried out (for example because there are too many random variables, i.e., too much uncertainty). Nowadays, MC methods are employed in many different fields, from optimisation [Robert and Casella, 2010] to decision making [Jiménez-Martín et al., 2005]. MC methods leverage the Central Limit Theorem [Johnson, 2004] as a main mathematical result. The theorem discusses the mean of a random variable with an arbitrary probability distribution, under the assumption that the variance of the distribution is finite. The theorem states that, if one draws infinitely many samples from the random variable, the distribution of the arithmetic mean of the samples asymptotically converges to a normal distribution, regardless of the original variable distribution.

The application of MC approaches allows to conduct an arbitrary number n of tests and measure the random variable X, obtaining a set of outcomes $\{x_1, \dots, x_n\}$. Then it is possible to determine the mean value $\bar{x}$ as the arithmetic average of the tests outputs,

$$\bar{x} = {}^{1}\!/_{n} \sum_{i=1}^{n} x_i. \tag{1}$$

The computed arithmetic mean $\bar{x}$ is also a random variable. The Central Limit Theorem guarantees that its distribution converges to a normal distribution for increasing n, i.e., $\bar{x} \sim \mathcal{N}(E[X], {}^{\sigma^2}\!/_{n})$, where $E[X]$ is the expected value of the random variable X and σ^2 is the variance of X. When n is big enough, the observed mean value converges to the actual expected value for the quantity of interest. This result is well-known in statistics and it holds irrespective of the specific software application under test. In fact, convergence is guaranteed independently from the probability distribution of the performance metric. However, there is no general result on the speed of the convergence and it is therefore application-dependant. With MC sampling, the significance of the test results and the choice of n is therefore left as an arbitrary choice to the testing engineer.

MC methods are therefore naturally used to evaluate the *average behaviour* of a system. By average behaviour $\bar{x}$ we mean a performance that best summarizes the different possible performances that the system can expose.[3] The standard deviation σ complements this information by quantifying instead the *spread* of the possible performance. By spread we mean the width of the range of possible performance values. The standard deviation can be estimated using the *sampled standard deviation*:

$$\bar{\sigma}^2 = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n-1}}. \tag{2}$$

Intuitively, since the sampled mean is a description of the average behaviour, each randomly generated test carries significant information about it. For this reason relatively few test can provide already a good convergence of the sampled mean $\bar{x}$ to the expected value $E[x]$. This is still application dependant and there is no general approach for quantifying this convergence.

Using MC methods we could be able to state, using again the recovery time example from Section 2, that *if the variance is sufficiently low, the system in is most likely recovering from a server failure in a time close to* 42 *seconds*. Unfortunately the theory doesn't allow us to rigorously quantify the "*most likely*" words used in the statement. In the same way, also the word "*close*" cannot be generally quantified rigorously. Finally, we cannot quantify the confidence that we can have in this statement. In fact, the confidence depends on whether the number of tests is sufficient to apply the central limit theorem or not.

MC methods have found limited use in the context of software testing [Korver, 1994; Singh and Pal, 2013]. None of these works focuses on the testing of self-adaptive software. In [Singh and Pal, 2013] MC methods are used to test the reliability of a software system, while [Korver, 1994] generally discusses how MC methods can be applied to software testing.

Extreme Value Theory (EVT): The Extreme Value Theory [Haan and Ferreira, 2010] (EVT) study a random variable around the tails of its distribution. This is opposed to MC methods that study the behaviour of a variable around its average. EVT could therefore be used when we specifically want to analyse the software's *worst-*

[3] Formally, the average corresponds to a normalized sum of the possible outcomes weighted by their probability. In general, the average value is not necessarily the most *probable* outcome, and it is not even guaranteed to be an actually *possible* outcome. For example, if we average the number of attempts necessary to obtain a response from a server, the average will likely have decimal values, but the possible outcome is only an integer. These aspects also limit the effectiveness of analyzing a performance metric by looking at its (sampled) average.

case behaviour, e.g., what is the maximum memory occupation of a program. The theory is nowadays widely adopted to study rare phenomena such as earthquakes, quantitative risks in finance, but also extreme events in engineering [Santinelli et al., 2014; Cazorla et al., 2013].

The role of the Central Limit Theorem for MC sampling is taken by the Fisher–Tippett–Gnedenko Theorem [Fisher, 1930] for the EVT. The Fisher–Tippett–Gnedenko theorem defines the family of distributions to which the maximum value of a set of samples converges. The family of distributions is called the Generalised Extreme Value Distribution (GEVD) [Haan and Ferreira, 2010]. To apply EVT, we can look at a set of data (in our case the performance parameters obtained from the test cases) and extract a set of samples that belong to the tail of the dataset – i.e., a set of *maxima*. We then fit the the GEVD to the extracted maxima. In this way, we can obtain a probability distribution for the extreme value of the performance metric that could be observed in future executions of the system.

There are different practices to extract the maxima from a dataset. The most common are: (i) the Block Maxima, and (ii) the Peaks Over Threshold. The former defines a partition of the dataset and extracts the maximum value from each subset, the latter takes all the values that exceed some predefined threshold. The difference between the two methods stems from the possibility of partitioning the dataset or not (for example in smaller sets of data – acquired with different software releases). When the data is naturally partitioned into smaller sets, the block maxima methods is preferred. Since in our case each data belongs to the same partition we use the peak over threshold method.

Using the obtained maxima we can estimate the parameters of the GEVD. This distribution has the following form:

$$f_{GEVD}(x) = \frac{1}{\sigma}\left[1+\left(\frac{\xi\,(x-\mu)}{\sigma}\right)^{-(\xi+1)/\xi}\right], \tag{3}$$

where μ is the location parameter, σ is the scale parameter and ξ is the shape parameter. The location parameter defines the starting point, the scale parameter defines the weight of the tail and the shape parameter defines the rate of decrease of the probability density.

With the obtained distribution we can evaluate a probabilistic bounds τ on the performance of the system, i.e. $P(x > \beta)$. To do so we need to account for the probability that a value is actually a maximum and the probability of that maximum being greater than the bound. This can be computed multiplying the two probabilities:

$$P(x > \beta) = P(x > \beta | x > \tau) \cdot P(x > \tau), \tag{4}$$

where τ is the threshold used for the selection of the maxima. The first term in the multiplication is equivalent to the cumulative probability distribution of the GEVD. The latter can be estimated as the ratio between the total number of performed tests and the number of test that resulted in one of the maxima:

$$P(x > \tau) \approx n_{maxima}/n_{tests}.$$

In this way, we can use EVT to obtain a probabilistic bound for the performance of the system under test. When applied to the example from Section 2, we would be able to provide guarantees in the form: *there is a* 1.5% *probability that the recovery time is worse than 50 seconds*. If needed, a different numerical bound can be given – apparently with a different associated probability. This is due to the fact that EVT allows us to retrieve a complete distribution for the maximum of the performance.

EVT presents similar limitations compared to MC approaches [Embrechts, 2000]. EVT uses an arbitrary number of samples from the distribution of interest. Moreover, the choice of which and how many samples can be considered as the maxima is also arbitrary. There are also no results on the rate of convergence of the samples to the Generalised Extreme Value Distribution. Conversely to the MC methods that talk about the average case EVT discusses the rare cases. For this reason few tests carry information about such cases. Therefore the convergence to the GEVD is generally slow. Finally, as MC, EVT requires finite variance of the parameter that is sampled.

4. Methodology

In this section, we describe our approach to obtain probabilistic guarantees and its theoretical underpinning.

4.1 Limitations of Traditional Statistics

In Section 3.2 we described the traditional tools from statistics that could be used to obtain probabilistic guarantees when testing self-adaptive software: MC and EVT. Both those methodologies suffer from limitations that make them inconvenient for analysing the results of the testing campaign – i.e. being used in place of the “Scenario theory” block in Figure 1. These limitations are: (i) arbitrary choice of testing parameters, (ii) unknown, case-dependent, testing confidence (or *testing adequacy*), and (iii) assumption that the variance of the measured quantity is finite.

MC and EVT use an arbitrary number of samples for the desired estimation. The MC approach assumes that the set of samples is large enough that the Central Limit Theorem holds [Robert and Casella, 2005], and the EVT similarly relies on the convergence of the maxima samples to the Extreme Value Distribution [Embrechts et al., 1997]. Unfortunately, in both the theories, there is no general way to define how many samples are needed to achieve convergence.

The impossibility of quantifying the convergence to the Gaussian and Extreme Value Distributions has another relevant implication for the testing problem. If the desired testing confidence is not reached, it is impossible to quantify how many extra tests are needed to reach it. In other words, we cannot know *a priori* how the confidence will change when performing one extra test.

Another assumption needed by both EVT and MC sampling is that the performance parameter has finite variance. In practice, this means that either the probability of it being infinite must be very low, or that the parameter can only take finite values. Suppose we are trying to assess the worst-case execution time of a software function. The presence of a bug could cause the processor to stall and the function to never terminate. As long as the occurrence of this bug is sporadic, it is possible to use EVT and MC to determine metrics on the execution time. However, if the bug is triggered more often, the (higher) probability of an infinite execution time would prevent us from applying EVT and MC methods. Some commonly used engineering solutions can enforce finite variance in given performance parameters. An example of this is the presence of timeouts. Introducing a timeout does not help overcoming the limitation. In fact, the test that resulted in a timeout does not provide a sample of the possible performance of the system (i.e., conveys less information than its number-based counterpart, resulting only in a 'timeout reached' outcome). Merging this information in the statistical evaluation is non-trivial, and could even be detrimental and hide behaviours of the system.

Our proposal overcomes these limitations by formulating the testing problem as an infinite optimisation problem and solving it using the *scenario theory*.

4.2 Scenario Theory for Software Testing

The *scenario approach* [Calafiore and Campi, 2006] was developed in the field of robust control [Francis and Khargonekar, 1995]. However, it is more generally applicable than control design. It provides a method to solve *infinite convex optimisation problems*. Infinite convex optimisation problems are a class of optimisation problems that appear frequently in robust control design. However, they are also classically found in other fields, such as decision making, finance, and management [Ramponi and Campi, 2018; Calafiore, 2013]. The contribution of this paper is the formulation of the testing problem with the scenario approach and the study of the results that can be obtained for self-adaptive software. We show how this allows us to overcome the research challenges from Section 1.

In our testing problem, we want to find bounds for a performance parameter of an adaptive system (i.e., of the software and a given adaptation strategy implemented on top of it). In general, finding a safe and very pessimistic bound on what the software can achieve is trivial. The interesting question is how much we can move this bound toward higher performance. This problem can be formulated as: we would like to maximise the value of the performance parameter that we can safely guarantee when using a given adaptation algorithm.

The evaluation of this performance bound can therefore be seen as an optimisation problem. Solving optimisation problems means finding the extreme value of a quantity, either the highest or the lowest possible. In the following sections we introduce optimisation problems, the scenario theory, and how they can be used to bound the performance of a self-adaptive software.

Optimisation Problems: Optimisation problems are defined by: (i) one or more decision variables, (ii) a cost function, and (iii) a set of constraints. The decision variables are the quantities we can choose and change. The cost function is the quantity we would like to maximise or minimise, and it should be a function of the decision variables. The constraints are statements about the decision variables that we want our final solution to satisfy. An example of a problem that can be formulated as an optimisation problem is the travelling salesman problem [Applegate et al., 2011]. A salesman needs to determine a route to visit a given number of cities, minimising the travelling distance. The decision variables are the segments to add to the path, the cost function is the total travelled distance, and the constraint is that all the cities in the given list are visited at least once.

In our proposed testing methodology the decision variable is the worst-case performance of the adaptation strategy (i.e. the best value of the performance metric that we can safely guarantee), the cost function is the worst-case performance itself, and each of the test outcomes is a constraint. The performance bound evaluation therefore becomes the following optimisation problem: *maximise the performance that can always be guaranteed, under the constraint that it cannot exceed what is experienced in the conducted tests.*

Being even more practical and using the web application example from Section 1, suppose we want to provide guarantees on its maximum response time thanks to the adaptation strategy. We conduct a certain number n of tests. Each test is composed of servicing 1000 requests in random execution instances of the overall system, and monitoring their response times. We record the average response times in the vector $\mathbf{r} = \{r_1, r_2, \ldots, r_n\}$. Where r_i is the average response time of the web application for the 1000 requests of the $i-th$ test. These values are constraints on what the software can achieve. We then take the maximum element of the vector as our worst-case performance metric, $w_{\max} = \max\{r_1, r_2, \ldots, r_n\}$. If we tested all the possible execution cases, we could then say that we guarantee that the response time will be lower than the maximum value $w_{\max}$. However, for self-adaptive software the set of possible execution cases is likely infinite.

Ideally, if we could perform an infinite number of tests, we would test the system in every possible situation. In this way, we could obtain an exact evaluation of the worst case behaviour of the system. In practice, this is apparently not achievable, and we have to rely on only a finite number of tests. Despite this, when the number of tests is sufficiently large, it will still provide significant information about the general case.

Infinite Optimisation Problems: If we cast our (ideal) testing problem into an optimisation problem, we would have infinite constraints (the infinite test cases). For our web application example this would mean performing an infinite number of tests and obtaining the real bound. Unfortunately, solving an optimisation problem with an infinite set of constraints is not always possible (or desirable). Similarly, in our testing problem, we cannot perform infinite tests.

Scenario theory: The scenario theory [Calafiore and Campi, 2006] addresses the problem of solving an infinite optimisation problem while accounting only for a finite number of the constraints. The theory provides probabilistic guarantees on the generality of the solution. The scenario approach is used to solve infinite optimisation problems. The approach is to transform the infinite-sized problem into a finite-sized problem by *randomly* sampling a finite number of constraints from the infinite set of possible ones. Then, it is possible to solve the optimisation problem accounting only for the finite set of sampled constraints. The scenario theory allows then to quantify the uncertainty and the guarantees that are lost by only considering the finite set.

In the testing of our web application, this corresponds to obtaining the probabilistic guarantee that the average response time is lower or equal to $w_{\max}$ in a high percentage of the cases. This means that, with high probability, the future executions of the web application would not result in a higher average response time, i.e., $\mathrm{P}(r_m \leq w_{\max} \mid m > n) = p_w \approx 1$.

Using the scenario theory, we can compute the probability p_w that the solution – computed using the finite set – does not satisfy all the constraints in the possibly infinite set. For our testing problem, this means that we evaluate the worst-case performance using only a finite number n of test results. We then compute the probability that the obtained worst-case value $w_{\max}$ holds for all of the infinite tests that we could possibly run – i.e., we compute the probability that in future tests we would obtain a worse value than $w_{\max}$, which is obtained using the first n tests, i.e., that $\exists m > n \mid w_m > w_{\max}$.

In our specific optimisation problem for testing, we have only one decision variable (the evaluation of our worst-case). We now state the relevant result of the scenario theory in that case.[4] We denote with ε the probability of *observing (in future executions) a performance value that is worse than the observed worst-case up to n tests* (i.e., $\varepsilon = 1 - p_w$). In the original optimisation framework this is the probability of not satisfying all the infinite constraints.

Using the scenario theory, we can evaluate the probability that, in our n test cases, we could have *missed a test case with a worse performance than the obtained bound.* We call this probability β and it is computed from ε and n as

$$(1 - \varepsilon)^n = \beta. \tag{5}$$

In the original optimisation problem, ε quantifies the probability that a new (randomly picked) constraint taken from the infinite set would invalidate the solution found using the finite set. In our testing analogy, ε is a quantification of how *tight* we want our bound to be. Choosing a lower probability ε means having a

[4] We omit the complete formula for an arbitrary number of decision variables, since it is not of interest in our case.

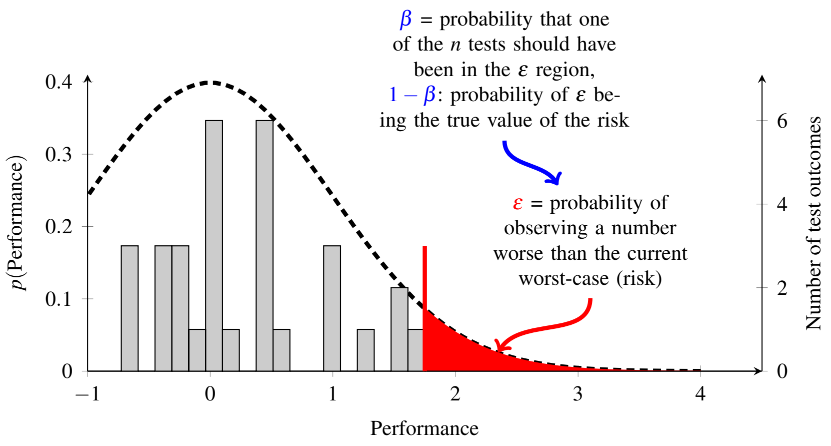


Figure 2. Graphical representation of the scenario parameters ε and β. In this case poor performance of the system is captured by high values of the performance parameter while low values correspond to more desirable performance. The histogram bars represent the performance observed in the test cases. The red bar is the observed word case. The red area corresponds to the probability ε quantified with ST of observing (in future executions) a performance worse than the worse performance observed so far. Figure from [Mandrioli and Maggio, 2020].

tighter bound, and choosing a higher value means that we allow for higher risk of not having observed the *true* worst-case. We remark that we can arbitrarily choose ε, but this will result in different degrees of confidence β that we can have in the obtained result.

The probability β can be seen as a quantification of how confident we are of our testing campaign result. A lower value of β implies that we are more confident and a higher value represents a higher probability that the final result is not correct. A tighter worst-case bound (lower ε), in fact, results in a higher β, a higher probability that we could have "missed" a relevant test case (constraint) in our sampling. In this sense, β can be seen as a *coverage* parameter, since it quantifies our confidence of having explored enough of the possible instances of the self-adaptive software behaviour.

Figure 2 shows a graphical interpretation of the probabilities ε and β. The dashed line shows the true and unknown probability distribution of the performance parameter. The histogram represents the observations that we obtained when measuring the performance of the system in our tests (i.e., our test results). The red bar indicates the worst case obtained during the testing campaign. The red area has size ε, i.e., ε is the probability that in the future we will experience a worst performance than the observed worst-case. Here, β is the probability that – assuming that ε is the correct area – we would not have observed a test case in the ε area during our

n observations. For example, if we had more test results, these could or could not be lower than the observed worst-case. In any case, with more observations, we are able to: (i) tighten the bound (i.e., decrease ε), (ii) increase the confidence (i.e., decrease β), or (iii) do both things to a lesser extent. Without running additional tests, we can tighten the bound at the cost of losing confidence in it. Alternatively, we could loosen the bound and increase our confidence.

We highlight that the theory does *not* require any prior knowledge on the probability distribution of the performance metric (i.e., on the dashed line in Figure 2). This is the strength of scenario theory with respect to the traditional methods that require assumptions on this probability distribution (e.g., its variance being finite).

We also remark that the test cases have to be randomly generated (or taken from the execution of the software in different scenarios). This is what guarantees the probabilistic characterisation. It could be argued, in fact, that what is actually used is only the test case where the system exposed the worst behaviour, and therefore this one is the only test case of interest. But identifying the testing conditions that expose the worst case might be not be straightforward and could require a greater effort than running a number of randomly generated test cases. In other cases, instead, the worst-case performance could be a trivial, arbitrarily bad performance. For example, the worst case response time of a web service will be infinite if all the servers become unavailable. Differently, we ask instead the following question: given a number of tests we ran on the real system, what is the average response time that we can guarantee in 99% of the cases? We argue that this probabilistic characterisation of self-adaptive software is (i) simpler to achieve and, (ii) more interesting than its deterministic counterpart. Therefore, when taking the probabilistic approach, even though a new test might not change the worst-case bound, it is still valuable because it increases the reliability and confidence in the obtained bound.

This probabilistic characterisation of the guarantees specifically addresses the research challenge **CH1**. Our argument is that, since deterministic guarantees cannot be given for adaptive systems, we should aim for probabilistic ones. Within the choice of probabilistic guarantees we have then addressed the other two research challenges. In fact, we have showed how to apply scenario theory for quantifying the system performance and testing confidence. Respectively, ε quantifies the probabilistic bound on the performance (**CH2**), and β quantifies the testing adequacy (**CH3**).

5. Experiments

This section aims at validating the proposed methodology. Our approach is designed to: (i) provide formal probabilistic guarantees from experiments (**CH1**), (ii) allow us to perform a fair comparison of different adaptation strategies (**CH2**), (iii) quantify the trade-off between the number (and cost) of experiments and the obtained probabilistic confidence (**CH3**). Finally, every method has to take into account the choice of test inputs. Hence, we explain how the choice of randomized testing inputs can affect the results of the testing campaign (**CH4**)

The proposed approach (shown in Figure 1) is *application independent*. We highlight this strength presenting experimental data from well-established adaptive software with different application domains: healthcare, video processing, and traffic flow optimisation.

In this section we show two different applications of the approach using the three presented analysis tools: MC, EVT, and ST. We highlight and discuss the respective limitations and what each technique can be used for.

In particular, in Section 5.1 we focus the discussion on the trade-off between the number of performed tests and the obtained probabilistic confidence using a simulation tool for the Tele Assistance Service (TAS) [Baresi et al., 2007; Weyns and Calinescu, 2015]. This shows how we address the research challenges **CH1** and **CH3**. In Section 5.2 we focus on the comparison of different adaptation strategies using the Self-Adaptive Video Encoder (SAVE) [Maggio et al., 2017b]. Our approach allows us to address the research challenge **CH2**. Furthermore we include a discussion on the possible consequences of a partition of the test inputs, this concerns the discussion the research challenge **CH4**. In Section 5.3 we use the TRAPP case study to discuss the role of test inputs definition in our testing methodology: this discussion addresses the research challenge **CH4**.

5.1 Data vs. Confidence Trade-Off

Aim: The aim of this case-study is to discuss the different probabilistic guarantees that can be provided with the different methodologies. To do so we show the complete application of the different methodologies (MC, EVT and ST) to the TAS system and comment the results. The results expose, among other facts, the limitations of the traditional statistical approaches in terms of confidence evaluation. Conversely, we discuss the probabilistic guarantees obtained with ST: the probabilistic performance bound and the testing confidence are discussed. More specifically, when using ST, we can offer guarantees on the software performance level, even for test cases that have not been explicitly executed (**CH1**), and we evidence the direct connection between the amount of collected experimental data and the probabilistic testing confidence (**CH3**).

Self-Adaptive Software: TAS is a service-oriented software application that provides care and assistance to elderly people that suffer from chronic diseases [Baresi et al., 2007]. The software [Weyns and Calinescu, 2015] periodically monitors patients conditions using sensors and activates a chain of services invocations. First, the patient conditions are sent to an *Analysis Service*, that inspects the data and determines the next steps to be taken for the patient well-being. The outcome of the analysis is one of the following: (i) do nothing, (ii) invoke a *Drug Service* that will compute a new medicine dosage, or (iii) invoke an *Alarm Service* that will dispatch an ambulance.

Each service can be realised by multiple service providers, potentially doing different computations that follow the same specification and interface. During the execution of the software, the selection of which provider to invoke to obtain a given functionality introduces an element of choice in the management of each request. Service providers have different properties; e.g., quality of the service, availability, success rate, and failure probability. In our experiments we focus on service rate and availability in the presence of failures, i.e., the number of requests processed per time unit and the probability of serving incoming request successfully.

The presence of different service providers and variety of potential needs for each request introduces the need to adapt the software behaviour to the current operating conditions. Adaptation strategies were introduced with the aim of selecting given services based on properties to be enforced for the overall system, e.g., [Shevtsov and Weyns, 2016; Caporuscio et al., 2017; Maggio et al., 2017a; Edwards and Bencomo, 2018]. In our experiments, the adaptation strategy should recognise the service providers with higher service rates and prioritise them when distributing the requests. Also, since services might not always be available, the adaptation layer should avoid submitting requests to unavailable service providers.

To identify the best choices, the adaptation layer stores one number per service provider, called *weight*. For all the alternatives, the weight is initialised to 1 and incremented or decremented (using a fixed step equal to 50 in our experiments) based on the service performance. For each successfully processed request, the weight increases, and for each failed invocation the weight decreases. We further introduce a timeout and reset the weight to 1 if the service invocation failed for all the requests sent in the timeout interval.

When distributing the requests, the weights are used to define a probability distribution over the different providers of a given service. The probability distribution can be obtained by normalising each of the weights over their overall sum. The requests are distributed according to this probability distribution. We limit the weights to an interval between 1 and 1000. This avoids that overly positive weights attract all the requests. In the same way, negative weights imply that the service provider is never chosen, making it impossible to recover even in case of potentially correct operation.

Test Design: We use the TAS case study to highlight the trade off between data and confidence, i.e., how the exploration of the system's behaviour improves with the increasing number of tests. The definition of which inputs should be randomised is critical for the correct coverage of the system's behaviour. Here, we randomise: (i) the requests profile, i.e., the number of incoming requests; (ii) the workload mix, i.e., the type of incoming requests; (iii) the availability of the different service providers, i.e., a provider being reachable or not; and (iv) the reliability of the service providers, i.e., request processing may fail due to internal reasons.

We can use one or more performance parameters, depending on the specific software and on what are the aspects that we want to test. The performance parameter

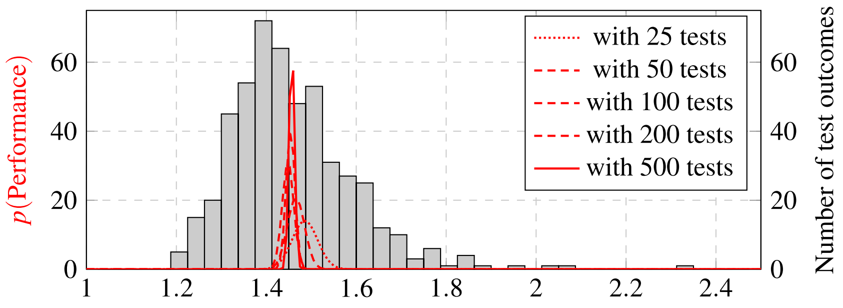


Figure 3. TAS: Histogram of experienced average number of attempts and plot of the obtained normal distribution of the sampled average performance.

should be representative of the behaviour of the adaptation layer. Practically, this means that it should enable the distinction of whether the adaptation layer worked well for the specific test case, or not. In the TAS case we want to build a system that is robust to the occurrence of failures. We choose as performance parameter the average number of attempts needed for a request to be correctly handled. Lower numbers indicate better adaptation, 1 being the best possible value (often not achievable).

Results MC: Figure 3 shows the histogram of the obtained worst case in the different tests. For each possible performance value on the horizontal axis the column above is proportional to the number of tests that exposed that performance. In the same figure, we also plot the gaussian distributions obtained for the sampled mean and its sampled variance using an increasing numbers of test cases.

We use this figure to investigate what MC methods allow us to state about the average performance of the system and comment on its applicability. The obtained gaussian distributions do not change significantly when increasing the number of tests – i.e., the different line plots are close to each other. This shows the quick convergence of the MC methods: already with 25 tests *we obtain some confidence that we can expect (on average) a performance of around* 1.45 *number of attempts with this adaptation strategy*.

On the other side, this evaluation does not give formal guarantees on such statement. Specifically, we would like to quantify the words "*around*" and "*some confidence*" from the statement above, but with MC this cannot be done in the general case. As an example, we cannot state that 1.45 is the performance that we are most likely to observe. The most probable performance, in fact, seems to be slightly lower according to the histogram.

To summarise, the experiments show that MC methods can be used to get a rough evaluation of the average behaviour of an adaptive system. Their quick convergence allows to achieve this with relatively few experiments. But, if the purpose

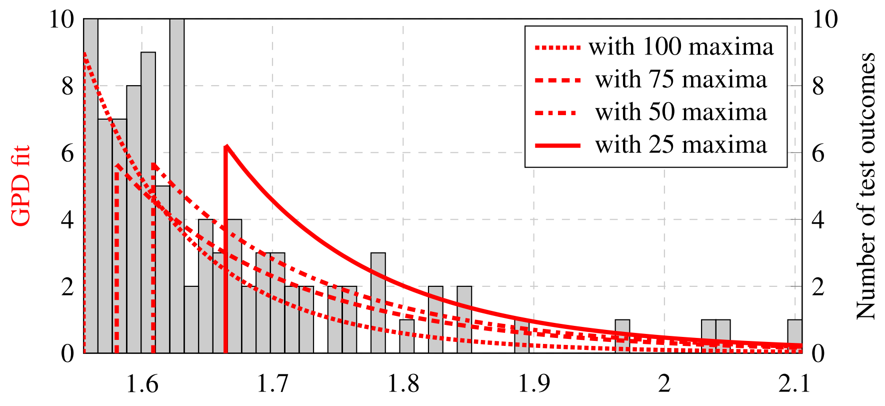


Figure 4. TAS: Histogram of all the test outcomes that exceed 1.5554 average attempts per request. In red are plotted the GEVDs fitted for different maxima choices in the TAS case study – namely 100, 75, 50, 25 maxima.

of the testing campaign is to formally constrain the adaptation performance, they lack of a general approach to the evaluation of the testing confidence.

Results EVT: Figure 4 shows the histogram of 100 maxima from the testset of the TAS system. Overlying to the histogram, are plotted the GEVDs fitted using different numbers of maxima, namely: 100, 75, 50, and 25. The application of EVT provides a full probabilistic characterization of what can possibly be the worst case performance of the system.

If we take one set of maxima and the associated fitted distribution we can provide probabilistic guarantees on what is the worst case performance of the adaptation strategy. For example if we take 1.9 as candidate worst case the fitted distributions can be used to compute the probability of obaining a performance worse than that. Using the distribution fitted to 100 maxima we would obtain *a* 0.92% *probability of obtaining a performance worse than* 1.9. The same statement could be made for different candidates for the worst case performance (apparently associated to a different probability).

Unfortunately, this evaluation doesn't include a quantification of the confidence we can have on the statement. This depends on whether the used maxima are sufficient to obtain convergence to the GEVD and the theory doesn't provide a way to quantify it. Experimental evidence of such statement is that the different choices of the maxima provide different results for the worst case probability. Using 75 maxima we would have in fact obtained a probability of 0.73%, using 50 maxima we would have obtained a probability of 0.58%, or using 25 maxima we would have in fact obtained a probability of 0.37%. The theory doesn't provide a way to evaluate the confidence and therefore to state which of those choices can be considered more

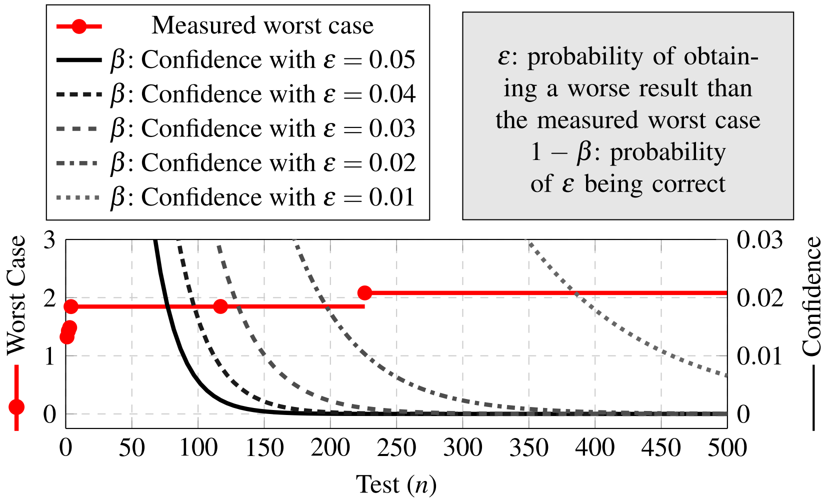


Figure 5. TAS: Measured worst case and confidence level varying the number of performed tests using the Scenario Theory. Figure from [Mandrioli and Maggio, 2020].

appropriate or reliable.

Results ST: Figure 5 shows the evolution of our quantities of interest when we perform an increasing number of tests and analyse them using ST. In particular, it shows: (i) the worst experienced average number of attempts needed per request (using the left y-axis), and (ii) the confidence β in the test outcome for different values of ε (using the right y-axis).

In the figure, we highlight with circle markers the newly experienced worst cases. The worst case is monotonically increasing with the number of conducted experiments. For example, in test #226, the average number of attempts per request to complete the TAS cycle is 2.081. This is a new worst case, as the previously experienced value was 1.8479 (from test #117).

The probability of not performing a relevant test (i.e., a test that would lead to a different worst case) is monotonically decreasing with the number of performed experiments. Analogously, a higher number of test cases is leading to a higher test coverage. Despite an unchanged worst case, between tests #117 and test #226, our confidence in the experimental results grew (lower values of β).

Decreasing the value of ε means being more conservative with our evaluation. The non-solid lines show the confidence β with smaller values of ε (up to 1%). Many more experiments are needed to obtain the same level of confidence when a smaller ε is selected.

The quantity β is the key difference between ST, and EVT or MC. Within ST, the test brings information both on the reliability of the testing process and the system under test itself. In MC and EVT the information carried by the tests is used only to evaluate the system's performance. Differently from MC and EVT the testing confidence allows the testing engineer to make a conscious choice on the number of randomly generated test cases according to the needed testing confidence and performance evaluation.

Using the scenario theory, we can state:

Based on the results of $n = 500$ tests, requests sent to TAS (with the described adaptation strategy) will not need more than 2.081 attempts on average to complete (despite service failures) with probability $1 - \varepsilon = 0.98$. This statement is correct with probability $1 - \beta = 0.99996$.

This performance is apparently strongly dependant on the chosen adaptation strategy. More interestingly, it does not depend on the specific values of the quantities that have been randomised for the test case generation. Conversely, we could determine the number of tests to be performed based on the desired ε and β values:

Given the desired probabilistic guarantees of confidence of $1 - \beta = 0.99996$ and a bound that holds in 98% of the cases, we perform $n = 500$ tests. In our case, the 500 tests indicate that in the worst case 2.081 attempts are needed on average per request.

Suppose that we could afford to conduct only $n = 250$ tests. In Figure 5 we can see that the measured worst case is the same as the complete test campaign. However, keeping $1 - \varepsilon = 0.98$, we could only claim a lower confidence in our test findings:

Based on the results of $n = 250$ tests, requests sent to TAS (with the described adaptation strategy) will not need more than 2.081 attempts on average to complete (despite service failures) with probability $1 - \varepsilon = 0.98$. This statement is correct with probability $1 - \beta = 0.9936$.

Vice versa, we could also determine the larger bound ε that we need to accept for if we wanted the same confidence $1 - \beta = 0.99996$ for 250 experiments. In this case we would obtain $1 - \varepsilon = 0.9603$.

5.2 Adaptation Strategies Comparison

Aim: The aim of this second set of experiments is to show the use of the proposed methodology for the comparison of different adaptation strategies. We run the tests and quantify the performance for each case with all the three discussed tools. The experiments expose the limitations of MC and EVT in enabling fair comparison. This is achieved, instead, with the use of ST (**CH2**). We aim at using the presented statistical tools to compare in a fair way the different adaptation strategies. Moreover, we also show the application of the scenario theory for testing with different and conflicting adaptation requirements (**CH1**). To further emphasise the validity of the proposed methodology, in this section we run the tests using the real software, rather than a simulation tool.

Self-Adaptive Software: SAVE [Maggio et al., 2017b] is a video encoding tool that aims at automatically achieving the desired size compression of a video stream whilst preserving as much as possible of its content. We target video broadcasting services, where multiple videos are streamed with a fixed amount of bandwidth and unpredictable demands. We also assume that the video content is not known a priori and is expected to change over time. The need for adaptation arises from the strong dependence of the encoding performance on the specific content of the video.

The adaptation strategy should leverage the frame characteristics to autonomously find an effective combination of encoding parameters. For each frame, the adaptation layer selects: (i) the *quality* parameter that specifies the compression density. It ranges between 1 and 100, where 100 preserves all frame details and 1 produces the highest compression; (ii) the *sharpen* parameter, which specifies the size of a sharpening filter to be applied to the image. The filter size ranges between 0 and 5 where 0 indicates no sharpening; (iii) *noise correction*, which specifies the size of a noise reduction filter, also between 0 and 5. High filtering should in general generate a more uniform image, making it simpler to compress.

For each frame the adaptation layer measures size and quality and selects the encoding parameters accordingly, using its own algorithm. The size is measured in bytes and the quality is measured using the Structural Similarity (SSIM) index [Zhou Wang et al., 2004]. This index is a unitless metric that ranges between 0 and 1 and quantifies the similarity between the original and the encoded frame (high index meaning high similarity). The measurements are used to evaluate the *size error* and the *SSIM error* as differences between the measured values and the desired ones.

We compare four different adaptation strategies, two from the original artifact [Maggio et al., 2017b] and two developed specifically for this work:

- **Random**: this adaptation strategy (from the original artifact) selects random encoding parameters. We use it as a baseline for our evaluation.
- **Model Predictive Control (MPC)**: this adaptation strategy (from the original artifact) exploits model predictive control algorithm [Garcia et al., 1989]. It solves a model-based optimisation problem for each frame and uses the result to determine the encoding parameters for the next frame. For our tests, we used the tuning parameters from the original publication [Maggio et al., 2017a].
- **Integral**: we developed an heuristic adaptation strategy, inspired by control theory principles. Here, the size error is used to choose the quality parameter. If the size is larger than the desired one, the quality parameter is reduced

Table 1. SSIM performance [adimensional].

	Mean	Var	Max	EVT (30%)	EVT (20%)	EVT (10%)
Random	0.0710	±0.0054	0.3251	0.002720	0.002602	0.002264
MPC	0.1145	±0.0068	0.4565	0.004850	0.005141	0.002810
Integral	0.0315	±0.0029	0.1685	0.003230	0.002876	0.001808
Greedy	0.0135	±0.0018	0.1777	0.002010	0.003062	0.003528

Table 2. Size performance [bytes].

	Mean	Var	Max	EVT (30%)	EVT (20%)	EVT (10%)
Random	8806	±1033	82488	0.003925	0.005366	0.004677
MPC	492	±94	8718	0.006511	0.005625	0.005046
Integral	126373	±13942	992342	0.006526	0.006001	0.005460
Greedy	1885	±318	35191	0.004277	0.002834	0.004337

by 5. If smaller, the quality is increased by 5. The SSIM index determines the choice of noise and sharpen filter radius. Both are increased by 1 if the quality is more than desired, and reduced otherwise. From an analytical perspective, the errors are *integrated* to perfect the encoding parameters choice.

- **ε-Greedy**: this adaptation strategy is based on the homonym machine-learning algorithm [Sutton and Barto, 1998]. More specifically it belongs to the class of reinforcement learning algorithms. It alternatively leverages two adaptation approaches: (i) a *greedy* approach that exploits the knowledge of the best parameters already encountered with probability $1-\varepsilon$, and (ii) a random approach that explores new possible choices, by randomly selecting new parameters with ε probability. The performance of a given choice of parameters is quantified based on the errors and normalised by the desired values. Higher similarity and lower size are desired, inducing errors that are close to zero. The greedy approach chooses the set of parameters that is associated to the lowest performance value. We use $\varepsilon = 0.2$.

Test Design: In SAVE, adaptation takes place along a stream of frames, i.e. the feedback from one frame is used to improve the encoding of the next frame. To capture the behaviour of the adaptation strategy, each test should be an adequately long video, in which changes occur, triggering the need for adaptation. We would like to evaluate the performance of the different adaptation strategies independently from the content of the processed videos.

According to the proposed methodology, we define a set of videos that can be considered a random sample, with respect to their content. Here, we used the *User Generated Content* dataset from Youtube [Wang et al., 2019]. This dataset is representative of videos uploaded by users to Youtube. The videos are classified in categories and we focused on the sport category, because, due to the ever-changing

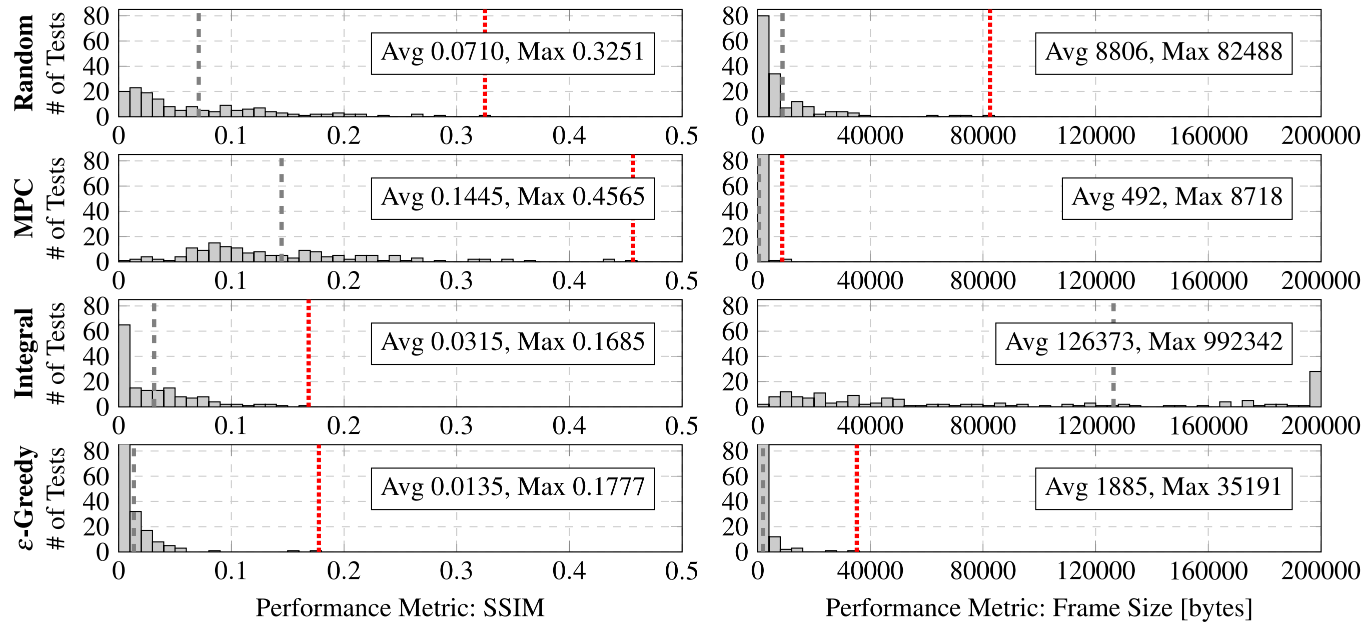


Figure 6. SAVE: performance of the adaptation over the Youtube dataset with different techniques: Random, MPC, Integral, and ε-Greedy. The tables report the sampled mean and variance, the worst case and the worst case probability computed with EVT using different numbers of maxima. The histograms show the performance observed in all test cases and highlight the mean (grey dashed line) and worst case (red dotted line). Figure from [Mandrioli and Maggio, 2020].

scene, these are usually the most difficult to encode for real-time streaming and will expose the most of the adaptation strategy properties. The database contains 160 sport videos.

The adaptation strategy tries to achieve multiple objectives (a given size of the encoded frames, and a given content loss) at the same time. To capture the results obtained for both objectives, we define two different performance parameters, used to measure the outcome of the tests. The encoding performance on a single frame is directly quantified as the errors on: (i) the encoding size and (ii) the SSIM. For performance evaluation, we only consider relevant the cases in which the size is larger than the desired value or the quality is lower than the setpoint.

Intuitively, the size error is a problem when the images require more bytes than desired, and the SSIM quality is a problem when the image has less information than desired. We therefore evaluate the performance over a video of an adaptation strategy as the average of the size and SSIM errors weighted with the REctified Linear Unit, $relu(\cdot)$ function. The $relu(\cdot)$ function returns 0 for negative inputs and leaves the input unchanged for positive values. The complete formula for the performance parameters is shown in Equation (6), where $SSIM_v$ and $SIZE_v$ are the integrated errors on the video v, $SSIM_{sp}$ and $SIZE_{sp}$ are respectively the SSIM and size setpoints, $SSIM_i$ and $SIZE_i$ are the SSIM and size of the i-th frame and n_f is the number of frames in the video.

$$\begin{aligned} SSIM_v &= (1/n_f) \cdot \textstyle\sum_i relu(SSIM_{sp} - SSIM_i), \\ SIZE_v &= (1/n_f) \cdot \textstyle\sum_i relu(SIZE_i - SIZE_{sp}). \end{aligned} \tag{6}$$

In our evaluation, we use a SSIM reference of 0.9, preserving most of the content in the videos, and a frame size reference of 70% of the size of a frame randomly picked from the uncompressed video. The choice of having per-video references for the size is driven by the strong dependence of the frame size on the specific video.

Results: We ran the 160 encoding tests with each adaptation strategy. For each video v, we computed the two performance parameters $SSIM_v$ and $SIZE_v$ defined in Equation (6). The histograms in Figure 6 show the results of the tests.[5] The dashed grey lines mark the average performance for both similarity index and size, and the red dotted lines highlight the worst case experienced during the tests.

Tables 1 and 2 respectively show different performance metrics for the SSIM and frame size. The two leftmost columns contain the sampled mean and variance, used by the MC analysis. The *Max* column displays the maximum values experienced for the parameters in the tests, relevant for the ST approach. The three rightmost columns show three different probabilities computed with the EVT method. These are the probabilities of obtaining a performance value worse than the worse experienced value in the tests. We specifically computed three EVT probabilities using the same threshold value (the maximum value experienced in the experiments).

[5] In the figure, we enforce the same scales for the axes to ease the comparison between the different plots. This results in hiding part of the plot of the size performance for the Integral strategy.

Table 3. Analysis of sampling bias for testing with SAVE. Frame size errors are normalised using the reference value, to allow a fair comparison between the different input sample sets.

	Video resolution	**360P**		**480P**		**720P**		**1080P**		**2160P**	
Strategy	Errors on	SSIM	Size	SSIM	Size	SSIM	Size	SSIM	Size	SSIM	Size
Random	Worst case	0.188	20468	0.325	22370	0.208	7743	0.262	82488	0.111	74245
	Average	0.070	1786	0.101	3084	0.058	3457	0.096	9418	0.031	25351
MPC	Worst case	0.250	6368	0.457	3044	0.312	799	0.435	8718	0.229	8577
	Average	0.142	254	0.177	209	0.129	165	0.186	603	0.091	1186
Integral	Worst case	0.139	44223	0.169	74377	0.098	173045	0.130	377724	0.067	992342
	Average	0.031	17714	0.044	27752	0.023	61233	0.046	162939	0.012	347691
Greedy	Worst case	0.035	846	0.178	13309	0.154	25153	0.054	35190	0.018	15967
	Average	0.011	265	0.023	1121	0.013	1487	0.017	2543	0.004	3852

The three probabilities correspond to the GEVD being fitted to respectively the 30%, 20%, and 10% largest values from the test outcomes. The chosen threshold value allows us to directly compare the results of the EVT approach with the ST probabilistic bound.

For what concerns the ST analysis, the number of performed tests $n = 160$ allows for the scenario parameters $\varepsilon = 0.03$ and $\beta = 0.008$. As for the TAS case study, this is not the only possible choice and a tighter bound could be traded for lower confidence (e.g. $\varepsilon = 0.01$ and $\beta = 0.04$) or vice versa (e.g. $\varepsilon = 0.05$ and $\beta = 0.0003$). Apparently, the two quantities hold equally for each of the tested adaptation strategies.

For the size performance, the MPC adaptation strategy vastly outperforms all the other strategies. This is achieved at the price of a SSIM adaptation performing worse than the Random strategy – i.e. the baseline. This is consistent with the adaptation objectives stated in the design of the strategy, where the size compression was considered the main objective [Maggio et al., 2017a]. This is equivalently observed by all the three alternative analysis techniques – i.e. comparing the first (MC) and third (EVT and ST) columns of the tables.

The Integral adaptation achieves the complementary result with respect to the MPC strategy. It presents good performance (among the strategies studied here) from the point of view of the SSIM but exposes the worse performance for what concerns the size. This can be attributed to the decoupled approach between the adaptation objectives pursued with this adaptation. Size and quality are not really decoupled (although the adaptation strategy treats them as such) and cannot effectively be treated separately.

When we compare the SSIM performance for the Integral and the ε-greedy strategies, the average and worst-case metrics are in slight disagreement. Whilst the former suggests a preference for the ε-greedy approach, the latter (the bare maximum) favours the Integral adaptation strategy. However, the tail of the histogram obtained with ε-greedy approach (Figure 6) seems lighter – i.e., less test cases performing “around and above” the performance value of 0.1. Intuitively, we would expect this to result in a higher probability of exceeding this bound. This is the probability that we computed with EVT in the right-most three columns of Table 1. Unfortunately, the value significantly depends on the number of maxima used for the GEVD fitting: if 30% of the values are considered maxima we should compare 0.32% for the integral adaptation and 0.2% for the ε-greedy strategy, otherwise 0.28% and 0.3%, or 0.18% and 0.35% if were respectively 20% and 10% of the values. Apparently, the conclusion on which is the best strategy will be different depending on the chosen number of maxima. Using EVT, we are not equipped with tools to select a number of maxima. Whilst these might seem minor variations in the probabilities, we recall that we are discussing probabilities of rare events (worst cases). Those probabilities are therefore intrinsically small and also minor variations can have high relative significance. This exposes one of the main limitations in applying the EVT to the testing of self-adaptive software. Conversely, ST assigns the same probability to the two adaptation strategies, leaving in some sense the final choice to the testing engineer. In this case, the sampled average from the MC method helps the testing engineer in choosing which adaptation strategy to favour. Despite this, it does not allow us to formally state that the ε-greedy approach outperforms the Integral strategy.

When simultaneously looking at both performance parameters, the machine-learning based approach achieves good performance. The SSIM performance is comparable to the one of the Integral adaptation and the size performance is in the order of the tens of kilobytes. This latter performance parameter can be considered small with respect to the biggest frames in the dataset, whose size is a few gigabytes. The ε-Greedy adaptation strategy proves therefore to be the best one at simultaneously achieving both adaptation objectives. This can be attributed to the exploration of the possible combinations of encoding parameters and the coupled feedback used for the two objectives.

Our testing methodology, when leveraging ST, guarantees that the comparison between the different adaptation strategies is fair. This is based on the rigorous quantification of the confidence we can have in obtained bounds. In particular, for the ε-Greedy algorithm, ST ensures that with a probability of $1-\varepsilon = 0.97$ we will not observe: (i) an error worse than 0.1777 for the SSIM performance parameter, and (ii) an error worse than 35191 Kb for the size performance parameter (see Equation 6 for the performance definitions). The confidence in our test campaign is of $1-\beta = 0.992$, meaning that there is little probability of the choice of the adaptation strategy being wrong. Conversely, using MC and EVT, such formal statements would not be possible. Finally, if there was a need to tighten the bound or increase the confidence in the test campaign, the scenario theory would directly provide the extra number of test cases needed.

We highlight the difference between worst-case and average-case metrics. Analysing the average case (as done with the MC approaches) for the results in Figure 6, one would conclude that the Random adaptation strategy actually performs more or less as well as the others. However, this is not at all true for the worst-case metrics, which clearly expose the trade-off between size and quality and the difference between having an adaptation strategy that targets one or both these quantities and picking the next frame configurations at random.

Results with different inputs choices: To conclude the discussion on the results obtained with SAVE, we would like to discuss the impact of the choice of input videos. The discussion belongs to a more general remark on the choice of representative samples for the random inputs to be provided to the testing machinery.

Suppose that the broadcast videos are acquired by surveillance cameras and that we have a set of cameras with given resolutions. Initially, we envision cameras with 360P, 480P, 720P and 2160P resolutions. We therefore test the adaptation strategies using a diverse set of videos but only with the mentioned resolutions and draw some conclusion on the worst case frame size and similarity errors. When our system expands, we want to introduce additional cameras, with a new resolution of 1080P. In the testing phase, we did not collect collected any data on videos with such a resolution. However, given that we tested higher and lower resolutions, we could think that our test results are valid nonetheless and apply also to the extended set. We could then use the measured worst case in our tests and assume this is (most likely) not going to be violated. However, the input set of videos was not representative of the data that we then experience from the actual system.

To show this, we partition our initial data set in different subset with the given resolutions. Table 3 shows the worst case and average case errors for the subsets of our initial input sample. Computing the worst case value excluding videos from the 1080P resolution shows that it was necessary to test for this specific resolution and the corresponding test output changed our worst case. In fact, if one looks at the column representing the 1080P resolution videos, the worst case errors for the size of the resulting frames is much higher than it is for the other videos. Excluding the 1080P videos from the dataset would result is a much more favourable worst-case, which is not representative of what would have happened had the test sample being complete. On the contrary, including these tests from the beginning (i.e., before our system expansion) would have resulted in a conservative value being computed for the first setup.

The same remarks apply to average performance metrics and similar considerations hold for the video quality metric (SSIM) when one removes the 480P subset.

This shows that there is indeed a need for a representative set of input videos to properly provide guarantees on the worst-case experienced values. This does not simply apply to the video streaming service, but to any system that is tested using statistical methods (including ST).

5.3 Test Input Definition

Aim: The aim of this case study is to discuss how to define test inputs in a randomized testing campaign. We introduce and analyse an adaptive software application for traffic flow optimisation [Gerostathopoulos and Pournaras, 2019]. We assume that the software is utilised to understand if a given traffic adaptation strategy (implemented in the original artifact) is beneficial or not. We formally define the objectives of the testing campaign and show how those map to the choice of which inputs to randomise and which inputs have to be fixed across the tests (**CH4**). Leveraging ST we define the number of tests that are needed according to the desired probabilistic guarantees on the software performance (**CH1**). We ran two rounds of experiments (with and without adaptation) to identify the potential benefits of enabling the adaptation strategy. We discuss the connection between the outcome of the testing campaign and the choices made for the input randomisation. This case study shows the practical applicability of the proposed testing approach to the performance testing and evaluation of an adaptation strategy.

Self-Adaptive Software: TRAPP [Gerostathopoulos and Pournaras, 2019] is a self-adaptive framework for decentralized traffic optimization. It is based on the microscopic traffic simulator SUMO [Lopez et al., 2018], and implements the interoperation with the decentralised combinatorial optimizer EPOS [Pournaras et al., 2018]. Within TRAPP, smart vehicles populate a simulated network of roads defined by the user. Vehicles can be introduced in the map in arbitrary points and will drive to reach a desired destination. Each vehicle produces different possible routes that it can take to reach its destination. The generated routes are associated to a specific cost that captures the preferences of the passengers. The routing options of every vehicle, together with their cost, are periodically collected. The EPOS optimiser is then executed to produce a route choice for each vehicle that accounts at the same time for both the desires of the car passengers and the overall efficiency of the road network. We extended the TRAPP artifact to enable repeated randomised testing.[6]

The traffic optimisation problem complexity grows exponentially with the number of cars, and achieving the global optimum in a general fashion is impossible. Moreover, the best approach to the optimisation problem depends on the current state of the network: e.g. the distribution of the cars in the streets, the specific destinations of the cars, and more. For this reason, adaptive approaches to the execution of the optimisation in EPOS have been proposed. The adaptation idea is to monitor in real-time the performance of the network and of the traffic flow optimisation. Leveraging this information, the EPOS optimisation can be adapted in order to improve the overall performance of the system. Among others, in TRAPP can be adapted the planning horizon, the planning fairness, or the agents selfishness.

In our case study, we assume a scenario in which the administration of a city wants to improve the city mobility by adopting the framework proposed by TRAPP. An adaptation strategy has been developed and needs to be tested in order to assess the potential benefit. More specifically, we consider the strategy *avoid-overloaded-streets* proposed in the original paper [Gerostathopoulos and Pournaras, 2019]. The idea is that the adaptive software layer monitors in real time which streets are closer to the limit of their capacity and adapts the EPOS optimisation so that those streets are avoided if possible.

Test Design: Analogously to the original paper [Gerostathopoulos and Pournaras, 2019], we quantify the performance of the adaptation strategy as the average trip overhead of the trips completed by all the cars. The trip overhead is defined as the ratio between the trip duration and the ideal trip duration that would be achieved in absence of other vehicles (hence if the vehicle was travelling always at the maximum allowed speed). A well performing adaptation strategy will be able to redirect the cars through the fastest route, thus reducing their traveling time and consequently the average trip overhead. Conversely, if the starting and end points of the cars are changed, the average trip is also likely to change. In this latter case, the performance change is related to the specific change in the testing scenario and not to

[6] The code used for this set of experiments can be found in this repository: `https://github.com/ManCla/TRAPP`.

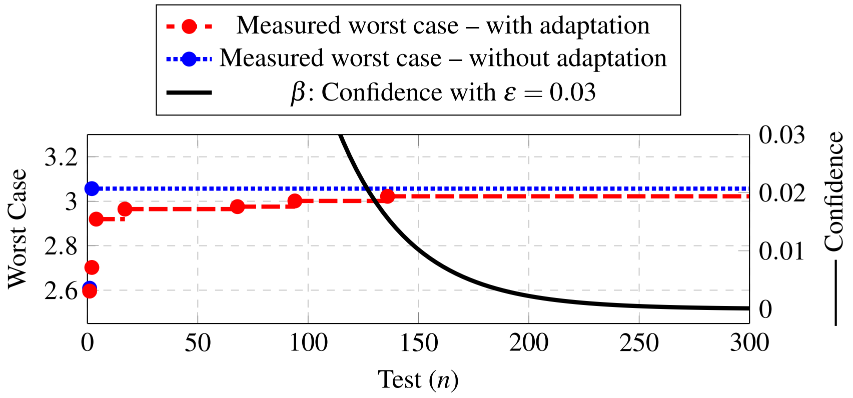


Figure 7. TRAPP: Measured worst case with adaptation stratergy (in red), without adaptation (in blue), and confidence level (in black) according to the desired 97% probabilistic bound computed using Scenario Theory.

the quality of the adaptation strategy. Leveraging the probabilistic approach taken in this paper, we therefore randomise the origin and destination of each trip over repeated tests. In this way, leveraging ST we obtain an evaluation of the adaptation performance that is independent of the source and destination parameters.

More in general, we depict a scenario in which the administration of the city mentioned above, wants a performance evaluation that is independent of both the specific vehicles and drivers that are currently populating the network, and also of the total number of vehicles in the streets. Following this specification, in our simulations we randomize these specific quantities. When a car is introduced in the network, we randomly pick values for its acceleration and deceleration (representing how performing are the engine and brakes of a car or driving style of the driver). Namely, for respectively the acceleration and deceleration we used two truncated normal distributions $\mathcal{N}(4,2)$ and $\mathcal{N}(6,2)$, both with unit of measure [kmh/s]. In order to avoid unrealistic car performances both values are forced to be at least 1. Said distributions represent a statistical knowledge of the cars used in the city. To account for the preferences of the different drivers, we select the weights for the different routing options from a uniform distribution between 0 and 1. The different routing options can optimise the length of the trip, the expected average speed, or a combination of the two. To achieve independence from the specific trips, the cars are introduced in random points in the network and are supposed to drive to equally random points. Finally, we assume that the city administration has an evaluation of the possible number of cars that populate the network. Specifically, it has been estimated that the number of cars can be any value between 800 and 1200 with equal probability. Therefore, the number of cars included in each test is chosen according to the estimated distribution.

Conversely, in the testing of the adaptation strategy, it will not be relevant to randomize parameters that will be fixed once the system is deployed. Examples of such parameters are: the network, the triggering period of the adaptation, and the duration of each test. While randomising those parameters will provide a more general evaluation of the chosen adaptation strategy, it will not provide further information concerning the actual use of the adaptive system. For example, it is not important that an adaptation strategy performs well independently from the specific city where the TRAPP framework is implemented. Once the system is deployed, the network is not expected to change significantly and require a consequent reaction of the adaptive system. For what concerns instead the triggering period, its analysis should be systematic rather than randomised, so that an optimal choice can be made in the system design. This kind of analysis is out of the scope of this paper and has been discussed in recent related research [Dorn et al., 2020], hence we chose the arbitrary period of 100 simulation ticks. The last mentioned parameter is the duration of the simulation. The choice of this parameter is driven by a trade-off between the efficiency and the relevance of the testing campaign. Apparently, efficiency purposes call for a simulation that is as short as possible. In our case study, we consider a test to have achieved significance when most cars have indicatively performed more than one trip. Since car trips can take from tens to hundreds of simulation ticks we chose for our tests a fixed value of 1000 simulation ticks.

In the depicted testing problem, we assume that a risk analysis requires that the obtained performance bound will hold in 97% of the cases. Equivalently, it is accepted a 3% probability that the adaptive system will not provide the expected performance: thus, for the application of ST ε is set to 0.03. Finally, we consider $1-\beta = 0.9999$ and acceptable confidence in the final result – i.e. the probability that ε is effectively equal to 0.03 and not larger. By applying Equation 5 we obtain the number $n = 300$ of required tests. In order to evaluate the effectiveness of the proposed framework, we ran two separate round of tests: one including the proposed strategy and another one where the EPOS optimisation is never executed.

Results: Figure 7, shows the worst case performance observed along the two rounds of tests together with the confidence increase (quantified using ST). In the figure, the blue plot shows the worst case observed without adaptation, the red plot shows instead the worst case in presence of the adaptation. The tests on the adapted system showed an overall worst case of 3.0223 for the average trip overhead. The tests on the system without adaptation showed an overall worst case of 3.0568. Under the light of the chosen scenario parameters, we can state that, *with a confidence of* 99.99%*, there is a* 3% *probability that a combination of the randomised parameters will lead to an average trip overhead larger than* 3.0223 *and* 3.0568 *for respectively the adapted and non adapted cases*. Conversely, the obtained bounds will not hold if different choices are made for the fixed parameters: the network, the adaptation triggering period, and the test duration. If these latter parameters are changed, the tests would have to be performed again in order to obtain bounds that are valid for the new set-up.

The experiments did not show a significant difference in the performance of the network when the adaptation was introduced. Hence, it can be concluded that the proposed adaptation strategy does not provide any relevant contribution to the traffic flow, and the city administration should investigate different solutions. In fact, given the choices that were made for the random parameters, there is a 97% probability that this conclusion will hold for any number of cars between 800 and 1200. On the other side, if a different map was to be chosen, the obtained values wouldn't hold anymore and the obtained conclusion would not be valid anymore.

6. Limitations and Validity Threats

In this section, we discuss validity threats to the proposed approach. Validity threats can be divided into *internal* and *external*. The methodology of the paper is a direct application of ST [Calafiore and Campi, 2006] and as such does not pose any internal validity threat. On the contrary, we identify external validity threats in how the test inputs are collected, how the scenarios are randomised, and how many sample data are available. These external validity threats result in three main limitations of the proposed approach.

The first one is rooted in the definition of the testing of an adaptive system. The need for adaptation in a system rises from limited knowledge of the operational environment. This generates an intrinsic limitation to the definition of test cases, since the software, as a requirement, should adapt to new unforeseen circumstances. On the other side the testing process is only as effective as the test cases are representative of the real use case. These two objectives of the introduction of adaptation and rigorous definition of test cases are colliding [Bahar et al., 2019]. The software engineer needs to synthesise a definition of the set of tests that adequately covers the adaptation use cases. However, the adaptive layer programmer has an interest in leaving the use cases as undefined as possible, for generality. In the TAS example, we would like the adaptation layer to handle general providers failures. However, this also means that (for proper testing) we need to define possible service failure patterns.

The second limitation arises from the interpretation of the performance parameters as random variables, and for this reason it is common to all the three statistical tools discussed. This interpretation is the key to exploit random sampling and to leverage the different theories that are based on probability theory. The roots of the limitation reside in the assumption of unbiased random sampling. Achieving unbiased random sampling can be challenging, especially when randomness cannot be quantified. The testing engineer must select a significant and relevant set of samples, e.g., sport videos with random content to test SAVE. The Scenario Theory reduces this limitation by not requiring any assumption on the probability distribution of the performance parameter. This allows to process data from tests that are conducted in a production environment (when available) and hence are, by definition, representative of the actual distributions.

A last limitation arises from the need to conduct many tests to achieve high confidence. EVT and ST are particularly affected by this, whilst MC seems to require a smaller number of tests – even though this cannot be generally guaranteed. Conducting many test cases can, in fact, be time-consuming and the process needs to be automated. On the other side, the number of needed tests is known a priory and allows for timely allocation of the resources. Also, within scenario theory the confidence grows exponentially with respect to the number of tests, avoiding the uncontrolled “explosion” of the number of tests to be executed.

7. Conclusions

In this paper we addressed the problem of testing the performance of a self-adaptive software system. Conventional testing techniques are limited in the guarantees they provide, due to the adaptation presence. The presence of adaptation makes this problem challenging, due to the need to test the system in the presence of uncertainty.

To deal with uncertainty, we investigated probabilistic techniques to analyse the resulting data. Moving to the probabilistic framework gave us the possibility of obtaining formal (albeit probabilistic) guarantees on the results of our testing campaign. We investigated classical statistical tools, like Monte Carlo Simulations and the Extreme Value Theory. In this investigation, we highlighted their limitations and shortcomings for the testing of adaptive software.

To overcome said limitations, we leveraged the scenario theory, a tool from robust control that was originally intended for the design of control systems in the presence of uncertainty. We reinterpreted the scenario theory results in light of our software testing problem. This allows us to provide formal probabilistic guarantees on the adaptation performance. Moreover, our method provides a probabilistic quantification of the testing adequacy, that can be used for the evaluation of testing coverage.

Finally, we empirically evaluated the effectiveness of our approach using three self-adaptive applications. We showed the trade-off between the experimental campaign volume and the confidence that can be obtained, demonstrated how to formally compare different adaptation strategies, and how to select randomised inputs for the testing process depending on the specified experimental evaluation objective. In our experimental results, we provided a thorough comparison of the application of Monte Carlo, Extreme Value Theory and the Scenario Theory. Our comparison showed why the latter is a better tool to test adaptive software.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH). The responsibility for the content remains with the authors.

References

Abu-Mostafa, Y. S., M. Magdon-Ismail, and H.-T. Lin (2012). *Learning From Data*. AMLBook. ISBN: 9781600490064.

Applegate, D., R. Bixby, V. Chvátal, and W. Cook (2011). *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press. ISBN: 9781400841103. URL: `https://books.google.se/books?id=zfIm94nNqPoC`.

Arcuri, A. and L. Briand (2011). "Adaptive random testing: an illusion of effectiveness?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. ACM, Toronto, Ontario, Canada, pp. 265–275. ISBN: 978-1-4503-0562-4. DOI: `10.1145/2001420.2001452`. URL: `http://doi.acm.org/10.1145/2001420.2001452`.

Arcuri, A., G. Fraser, and J. P. Galeotti (2014). "Automated unit test generation for classes with environment dependencies". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. ACM, Vasteras, Sweden, pp. 79–90. ISBN: 978-1-4503-3013-8. DOI: `10.1145/2642937.2642986`. URL: `http://doi.acm.org/10.1145/2642937.2642986`.

Bahar, R. I., U. Karpuzcu, and S. Misailovic (2019). "Special session: does approximation make testing harder (or easier)?" In: *2019 IEEE 37th VLSI Test Symposium (VTS)*, pp. 1–9. DOI: `10.1109/VTS.2019.8758649`.

Baresi, L., D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini (2007). "Validation of web service compositions". *IET Software* **1**:6, pp. 219–232. ISSN: 1751-8806. DOI: `10.1049/iet-sen:20070027`.

Bertolino, A. and P. Inverardi (2019). "Changing software in a changing world: how to test in presence of variability, adaptation and evolution?" In: Beek, M. H. ter et al. (Eds.). *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*. Springer International Publishing, Cham, pp. 56–66. ISBN: 978-3-030-30985-5. DOI: `10.1007/978-3-030-30985-5_5`. URL: `https://doi.org/10.1007/978-3-030-30985-5_5`.

Bertolino, A., P. Inverardi, and H. Muccini (2003). “Formal methods in testing software architectures”. In: Bernardo, M. et al. (Eds.). *Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003. Advanced Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–147. ISBN: 978-3-540-39800-4. DOI: 10.1007/978-3-540-39800-4_7. URL: https://doi.org/10.1007/978-3-540-39800-4_7.

Böhme, M. (2019). “Assurance in software testing: a roadmap”. In: *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER ’19. IEEE Press, Montreal, Quebec, Canada, pp. 5–8. DOI: 10.1109/ICSE-NIER.2019.00010. URL: https://doi.org/10.1109/ICSE-NIER.2019.00010.

Briand, L., S. Nejati, M. Sabetzadeh, and D. 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. ACM, Austin, Texas, pp. 789–792. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889212. URL: http://doi.acm.org/10.1145/2889160.2889212.

Calafiore, G. C. and M. C. Campi (2006). “The scenario approach to robust control design”. *IEEE Transactions on Automatic Control* **51**:5, pp. 742–753. ISSN: 0018-9286. DOI: 10.1109/TAC.2006.875041.

Calafiore, G. C. (2013). “Direct data-driven portfolio optimization with guaranteed shortfall probability”. *Automatica* **49**:2, pp. 370–380. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2012.11.012. URL: http://www.sciencedirect.com/science/article/pii/S0005109812005481.

Canfora, G. and M. Di Penta (2006). “Testing services and service-centric systems: challenges and opportunities”. *IT Professional* **8**:2, pp. 10–17. ISSN: 1520-9202. DOI: 10.1109/MITP.2006.51.

Caporuscio, M., R. Mirandola, and C. Trubiani (2017). “Building design-time and run-time knowledge for qos-based component assembly”. *Softw. Pract. Exper.* **47**:12, pp. 1905–1922. ISSN: 0038-0644. DOI: 10.1002/spe.2502. URL: https://doi.org/10.1002/spe.2502.

Cazorla, F. J., T. Vardanega, E. Quiñones, and J. Abella (2013). “Upper-bounding Program Execution Time with Extreme Value Theory”. In: Maiza, C. (Ed.). *13th International Workshop on Worst-Case Execution Time Analysis*. Vol. 30. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 64–76. ISBN: 978-3-939897-54-5. DOI: 10.4230/OASIcs.WCET.2013.64. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4123.

Chen, T. Y., F.-C. Kuo, R. G. Merkel, and T. H. Tse (2010). “Adaptive random testing: the art of test case diversity”. *J. Syst. Softw.* **83**:1, pp. 60–66. ISSN: 0164-1212. DOI: `10.1016/j.jss.2009.02.022`. URL: `http://dx.doi.org/10.1016/j.jss.2009.02.022`.

Cheng, B. H., R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle (2009). “Software engineering for self-adaptive systems”. In: Cheng, B. H. et al. (Eds.). *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Springer-Verlag, Berlin, Heidelberg, pp. 1–26.

Cheng, B. H. C., K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas (2014). “Using models at runtime to address assurance for self-adaptive systems”. In: Bencomo, N. et al. (Eds.). *Models@run.time: Foundations, Applications, and Roadmaps*. Springer International Publishing, Cham, pp. 101–136. ISBN: 978-3-319-08915-7. DOI: `10.1007/978-3-319-08915-7_4`. URL: `https://doi.org/10.1007/978-3-319-08915-7_4`.

DIppolito, N., V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel (2014). “Hope for the best, prepare for the worst: multi-tier control for adaptive systems”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Association for Computing Machinery, Hyderabad, India, pp. 688–699. ISBN: 9781450327565. DOI: `10.1145/2568225.2568264`. URL: `https://doi.org/10.1145/2568225.2568264`.

Dorn, J., S. Apel, and N. Siegmund (2020). “Mastering uncertainty in performance estimations of configurable software systems”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’20. Association for Computing Machinery, Virtual Event, Australia, pp. 684–696. ISBN: 9781450367684. DOI: `10.1145/3324884.3416620`. URL: `https://doi.org/10.1145/3324884.3416620`.

Dutta, S., O. Legunsen, Z. Huang, and S. Misailovic (2018). “Testing probabilistic programming systems”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Association for Computing Machinery, Lake Buena Vista, FL, USA, pp. 574–586. ISBN: 9781450355735. DOI: `10.1145/3236024.3236057`. URL: `https://doi.org/10.1145/3236024.3236057`.

Dutta, S., W. Zhang, Z. Huang, and S. Misailovic (2019). “Storm: program reduction for testing and debugging probabilistic programming systems”. 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, Tallinn, Estonia, pp. 729–739. ISBN: 9781450355728. DOI: 10.1145/3338906.3338972. URL: https://doi.org/10.1145/3338906.3338972.

Edwards, R. and N. Bencomo (2018). "Desire: further understanding nuances of degrees of satisfaction of non-functional requirements trade-off". In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '18. ACM, Gothenburg, Sweden, pp. 12–18. ISBN: 978-1-4503-5715-9. DOI: 10.1145/3194133.3194142. URL: http://doi.acm.org/10.1145/3194133.3194142.

Embrechts, P. (2000). "Extreme value theory: potential and limitations as an integrated risk management tool". *Derivatives Use, Trading and Regulation* **6**.

Embrechts, P., T. Mikosch, and C. Klüppelberg (1997). *Modelling Extremal Events: For Insurance and Finance*. Springer-Verlag, Berlin, Heidelberg. ISBN: 3540609318.

Ferrari, F. C., J. Noppen, R. Chitchyan, and A. R. Lancaster (2011). "Investigating testing approaches for dynamically adaptive systems work in progress". In: *Environment*.

Filieri, A., C. Ghezzi, A. Leva, and M. Maggio (2011). "Self-adaptive software meets control theory: a preliminary approach supporting reliability requirements". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, Lawrence, KS, USA, pp. 283–292. DOI: 10.1109/ASE.2011.6100064.

Filieri, A., H. Hoffmann, and M. Maggio (2014). "Automated design of self-adaptive software with control-theoretical formal guarantees". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE. ACM, Hyderabad, India, pp. 299–310. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568272. URL: http://doi.acm.org/10.1145/2568225.2568272.

Fisher, R. (1930). *The Genetical Theory of Natural Selection*. OUP Oxford.

Francis, B. and P. Khargonekar (1995). *Robust control theory*. The IMA volumes in mathematics and its applications. Springer-Verlag. ISBN: 9780387944432. URL: https://books.google.se/books?id=81vvAAAAMAAJ.

Garcia, C. E., D. M. Prett, and M. Morari (1989). "Model predictive control: theory and practice&mdash;a survey". *Automatica* **25**:3, pp. 335–348. ISSN: 0005-1098. DOI: 10.1016/0005-1098(89)90002-2. URL: http://dx.doi.org/10.1016/0005-1098(89)90002-2.

Gerostathopoulos, I. and E. Pournaras (2019). "Trapped in traffic? a self-adaptive framework for decentralized traffic optimization". In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 32–38. DOI: 10.1109/SEAMS.2019.00014.

González, C. A., M. Varmazyar, S. Nejati, L. C. Briand, and Y. Isasi (2018). “Enabling model testing of cyber-physical systems”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’18. ACM, Copenhagen, Denmark, pp. 176–186. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239409. URL: http://doi.acm.org/10.1145/3239372.3239409.

Gulisano, V., A. V. Papadopoulos, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas (2017). “Performance modeling of stream joins”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS ’17. ACM, Barcelona, Spain, pp. 191–202. ISBN: 978-1-4503-5065-5. DOI: 10.1145/3093742.3093923. URL: http://doi.acm.org/10.1145/3093742.3093923.

Haan, L. de and A. Ferreira (2010). *Extreme Value Theory: An Introduction (Springer Series in Operations Research and Financial Engineering)*. 1st Edition. Springer. ISBN: 144192020X.

Hänsel, J., T. Vogel, and H. Giese (2015). “A testing scheme for self-adaptive software systems with architectural runtime models”. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pp. 134–139. DOI: 10.1109/SASOW.2015.27.

Hervieu, A., B. Baudry, and A. Gotlieb (2012). “Managing execution environment variability during software testing: an industrial experience”. In: Nielsen, B. et al. (Eds.). *Testing Software and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 24–38. ISBN: 978-3-642-34691-0. DOI: 10.1007/978-3-642-34691-0_4.

Hierons, R. M. and M. G. Merayo (2009). “Mutation testing from probabilistic and stochastic finite state machines”. *J. Syst. Softw.* **82**:11, pp. 1804–1818. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.030. URL: http://dx.doi.org/10.1016/j.jss.2009.06.030.

Hwang, S.-Y., H. Wang, J. Tang, and J. Srivastava (2007). “A probabilistic approach to modeling and estimating the qos of web-services-based workflows”. *Inf. Sci.* **177**:23, pp. 5484–5503. ISSN: 0020-0255. DOI: 10.1016/j.ins.2007.07.011. URL: https://doi.org/10.1016/j.ins.2007.07.011.

Jiménez-Martín, A., A. Mateos, and S. Ríos-Insua (2005). “Monte carlo simulation techniques in a decision support system for group decision making”. *Group Decision and Negotiation* **14**, pp. 109–130. DOI: 10.1007/s10726-005-2406-9.

Johnson, O. (2004). *Information Theory and the Central Limit Theorem*. Imperial College Press. ISBN: 9781860945373. URL: https://books.google.se/books?id=r5XI8a0lYykC.

Joshi, K., V. Fernando, and S. Misailovic (2019). “Statistical algorithmic profiling for randomized approximate programs”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE 19. IEEE Press, Montreal, Quebec, Canada, pp. 608–618. DOI: `10.1109/ICSE.2019.00071`. URL: `https://doi.org/10.1109/ICSE.2019.00071`.

Korver, B. (1994). *The monte carlo method and software reliability theory.*

Lopez, P. A., M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. WieSSner (2018). “Microscopic traffic simulation using sumo”. In: *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE. URL: `https://elib.dlr.de/124092/`.

Maggio, M., A. V. Papadopoulos, A. Filieri, and H. Hoffmann (2017a). “Automated control of multiple software goals using multiple actuators”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. ACM, Paderborn, Germany, pp. 373–384. ISBN: 978-1-4503-5105-8. DOI: `10.1145/3106237.3106247`. URL: `http://doi.acm.org/10.1145/3106237.3106247`.

Maggio, M., A. V. Papadopoulos, A. Filieri, and H. Hoffmann (2017b). “Self-adaptive video encoder: comparison of multiple adaptation strategies made simple”. In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’17. IEEE Press, Buenos Aires, Argentina, pp. 123–128. ISBN: 978-1-5386-1550-8. DOI: `10.1109/SEAMS.2017.16`. URL: `https://doi.org/10.1109/SEAMS.2017.16`.

Mandrioli, C. and M. Maggio (2020). “Testing self-adaptive software with probabilistic guarantees on performance metrics”. In: *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM. ISBN: 9781450370431. DOI: `10.1145/3368089.3409685`.

Martina Maggio, C. (2020). *Artifact esec/fse 2020*. DOI: `10.5281/ZENODO.3896795`. URL: `https://zenodo.org/record/3896795`.

Mehmood, M. A., M. N. A. Khan, and W. Afzal (2018). “Automating test data generation for testing context-aware applications”. In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 104–108. DOI: `10.1109/ICSESS.2018.8663920`.

Micskei, Z., Z. Szatmári, J. Oláh, and I. Majzik (2012). “A concept for testing robustness and safety of the context-aware behaviour of autonomous systems”. In: *Proceedings of the 6th KES International Conference on Agent and Multi-Agent Systems: Technologies and Applications*. KES-AMSTA12. Springer-Verlag, Dubrovnik, Croatia, pp. 504–513. ISBN: 9783642309465. DOI: `10.1007/978-3-642-30947-2_55`. URL: `https://doi.org/10.1007/978-3-642-30947-2_55`.

Moreno, G. A., J. Cámara, D. Garlan, and B. Schmerl (2015). "Proactive self-adaptation under uncertainty: a probabilistic model checking approach". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. ACM, Bergamo, Italy, pp. 1–12. ISBN: 978-1-4503-3675-8. DOI: `10.1145/2786805.2786853`.

Moreno, G. A., A. V. Papadopoulos, K. Angelopoulos, J. Cámara, and B. Schmerl (2017). "Comparing model-based predictive approaches to self-adaptation: cobra and pla". In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '17. IEEE Press, Buenos Aires, Argentina, pp. 42–53. ISBN: 978-1-5386-1550-8. DOI: `10.1109/SEAMS.2017.2`. URL: `https://doi.org/10.1109/SEAMS.2017.2`.

Munoz, F. and B. Baudry (2009). "Artificial table testing dynamically adaptive systems". *CoRR* **abs/0903.0914**. arXiv: `0903.0914`. URL: `http://arxiv.org/abs/0903.0914`.

Oliveira Neves, V. de, A. Bertolino, G. De Angelis, and L. Garcés (2018). "Do we need new strategies for testing systems-of-systems?" In: *Proceedings of the 6th International Workshop on Software Engineering for Systems-of-Systems*. SESoS '18. ACM, Gothenburg, Sweden, pp. 29–32. ISBN: 978-1-4503-5747-0. DOI: `10.1145/3194754.3194758`. URL: `http://doi.acm.org/10.1145/3194754.3194758`.

Pournaras, E., P. Pilgerstorfer, and T. Asikis (2018). "Decentralized collective learning for self-managed sharing economies". *ACM Trans. Auton. Adapt. Syst.* **13**:2. ISSN: 1556-4665. DOI: `10.1145/3277668`. URL: `https://doi.org/10.1145/3277668`.

Qin, Y., C. Xu, P. Yu, and J. Lu (2016). "Sit: sampling-based interactive testing for self-adaptive apps". *Journal of Systems and Software* **120**, pp. 70–88. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2016.07.002`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121216301029`.

Ramponi, F. A. and M. C. Campi (2018). "Expected shortfall: heuristics and certificates". *European Journal of Operational Research* **267**:3, pp. 1003–1013. ISSN: 0377-2217. DOI: `https://doi.org/10.1016/j.ejor.2017.11.022`. URL: `http://www.sciencedirect.com/science/article/pii/S0377221717310330`.

Reichstaller, A. and A. Knapp (2018). "Risk-based testing of self-adaptive systems using run-time predictions". In: *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 80–89. DOI: `10.1109/SASO.2018.00019`.

Robert, C. P. and G. Casella (2005). *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag, Berlin, Heidelberg. ISBN: 0387212396.

Robert, C. P. and G. Casella (2010). “Monte carlo optimization”. In: *Introducing Monte Carlo Methods with R*. Springer New York, New York, NY, pp. 125–165. ISBN: 978-1-4419-1576-4. DOI: `10.1007/978-1-4419-1576-4_5`. URL: `https://doi.org/10.1007/978-1-4419-1576-4_5`.

Rosario, S., A. Benveniste, S. Haar, and C. Jard (2008). “Probabilistic qos and soft contracts for transaction-based web services orchestrations”. *IEEE Transactions on Services Computing* **1**:4, pp. 187–200. ISSN: 1939-1374. DOI: `10.1109/TSC.2008.17`.

Salehie, M. and L. Tahvildari (2009). “Self-adaptive software: landscape and research challenges”. *ACM Trans. Auton. Adapt. Syst.* **4**:2. ISSN: 1556-4665. DOI: `10.1145/1516533.1516538`. URL: `https://doi.org/10.1145/1516533.1516538`.

Santinelli, L., J. Morio, G. Dufour, and D. Jacquemart (2014). “On the sustainability of the extreme value theory for wcet estimation”. In: *OpenAccess Series in Informatics*. Vol. 39. DOI: `10.4230/OASIcs.WCET.2014.21`.

Santos, I. d. S. (2017). *TESTDAS: Testing MEthod for Dynamically Adaptive Systems*. PhD thesis. Universisdade Federal do Ceara, Fortaleza, Brazil.

Shevtsov, S. and D. Weyns (2016). “Keep it simplex: satisfying multiple goals with guarantees in control-based self-adaptive systems”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, Seattle, WA, USA, pp. 229–241. ISBN: 978-1-4503-4218-6. DOI: `10.1145/2950290.2950301`. URL: `http://doi.acm.org/10.1145/2950290.2950301`.

Singh, H. and P. Pal (2013). “Software reliability testing using monte carlo methods”. *International Journal of Computer Applications* **69**, pp. 41–44. DOI: `10.5120/11834-7554`.

Siqueira, B. R., F. C. Ferrari, M. A. Serikawa, R. Menotti, and V. V. de Camargo (2016). “Characterisation of challenges for testing of adaptive systems”. In: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*. SAST. Association for Computing Machinery, Maringa, Parana, Brazil. ISBN: 9781450347662. DOI: `10.1145/2993288.2993294`. URL: `https://doi.org/10.1145/2993288.2993294`.

Sutton, R. S. and A. G. Barto (1998). *Introduction to Reinforcement Learning*. 1st. MIT Press, Cambridge, MA, USA. ISBN: 0262193981.

Tramontana, P., D. Amalfitano, N. Amatucci, A. Memon, and A. R. Fasolino (2019). “Developing and evaluating objective termination criteria for random testing”. *ACM Trans. Softw. Eng. Methodol.* **28**:3, 17:1–17:52. ISSN: 1049-331X. DOI: `10.1145/3339836`. URL: `http://doi.acm.org/10.1145/3339836`.

Trubiani, C. and S. Apel (2019). "Plus: performance learning for uncertainty of software". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 77–80. DOI: 10.1109/ICSE-NIER.2019.00028.

Tse, T., S.-S. Yau, W. Chan, H. Lu, and T. Chen (2004). "Testing context-sensitive middleware-based software applications". English (US). In: *Proceedings - International Computer Software and Applications Conference*. Vol. 1, pp. 458–466.

Wang, H., W. K. Chan, and T. H. Tse (2014). "Improving the effectiveness of testing pervasive software via context diversity". *ACM Trans. Auton. Adapt. Syst.* **9**:2. ISSN: 1556-4665. DOI: 10.1145/2620000. URL: https://doi.org/10.1145/2620000.

Wang, Y., S. Inguva, and B. Adsumilli (2019). *Youtube ugc dataset for video compression research*. arXiv: 1904.06457 [cs.MM]. URL: https://media.withyoutube.com/.

Welsh, K. and P. Sawyer (2010). "Managing testing complexity in dynamically adaptive systems: a model-driven approach". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 290–298. DOI: 10.1109/ICSTW.2010.57.

Weyns, D. (2012). "Towards an integrated approach for validating qualities of self-adaptive systems". In: *Proceedings of the Ninth International Workshop on Dynamic Analysis*. WODA 2012. Association for Computing Machinery, Minneapolis, MN, USA, pp. 24–29. ISBN: 9781450314558. DOI: 10.1145/2338966.2336803. URL: https://doi.org/10.1145/2338966.2336803.

Weyns, D. and R. Calinescu (2015). "Tele assistance: a self-adaptive service-based system examplar". In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. IEEE Press, Florence, Italy, pp. 88–92. URL: http://dl.acm.org/citation.cfm?id=2821357.2821373.

Yatoh, K., K. Sakamoto, F. Ishikawa, and S. Honiden (2015). "Feedback-controlled random test generation". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Association for Computing Machinery, Baltimore MD USA, pp. 316–326. ISBN: 9781450336208. DOI: 10.1145/2771783.2771805. URL: https://doi.org/10.1145/2771783.2771805.

Yu, L., W. T. Tsai, Y. Jiang, and J. Gao (2014). "Generating test cases for context-aware applications using bigraphs". In: *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pp. 137–146. DOI: 10.1109/SERE.2014.27.

Zeller, A., R. Gopinath, M. Böhme, G. Fraser, and C. Holler (2019). “The fuzzing book”. In: *The Fuzzing Book*. Retrieved 2019-09-09 16:42:54+02:00. Saarland University. URL: `https://www.fuzzingbook.org/`.

Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli (2004). “Image quality assessment: from error visibility to structural similarity”. *IEEE Transactions on Image Processing* **13**:4, pp. 600–612. ISSN: 1057-7149. DOI: `10.1109/TIP.2003.819861`.

Paper II

Testing Abstractions for Cyber-Physical Control Systems

Claudio Mandrioli, Max Nyberg Carlsson, Martina Maggio

Abstract

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 process-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.

Under Submission to Transactions of Software Engineering and Methodology.

1. Introduction

Control is at the core of many Cyber-Physical Systems (CPS) and pervasive in modern life [Müller, 2017]. Control is found in many devices, from small consumer electronics like phones, to cars, or space vehicles [Bach et al., 2017; Molzahn et al., 2017; Menghi et al., 2019b]. Historically, control systems were built using mechanical devices, like hydraulic circuits [Maxwell, 2011]. Nowadays, they are built using software that interacts with the physical world through sensors and actuators [Åström and Wittenmark, 2013]. The ubiquity and criticality of software-based control systems makes their verification and validation of primary importance [van der Knijff, 2014; Briand et al., 2016].

A controller comprises of sensors, actuators, hardware and software, and is used to make a physical system behave according to given requirements [Åström and Wittenmark, 2013]. The union of controller and physical process is called "control system". A prominent example of control system is the cruise control of a car. Its objective is to ensure that the car reaches and maintains the desired velocity [Nilsson et al., 2016]. 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. This iteration of sensing and actuation creates a *closed loop* between the physical process 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 [Broy et al., 2007; Grässler et al., 2020]. This makes control software complex, and prone to errors. Unsurprisingly, control software requires a long and costly verification and validation process [Garca et al., 2020].

During the verification and validation process, engineers spend most time on testing [Zheng et al., 2017; Garca et al., 2020; Bertolino et al., 2021]. The main difficulty in testing control systems 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 [Menghi et al., 2019a; Afzal et al., 2020]. 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 [Maia et al., 2019]. The choice of which components to abstract defines different testing setups [Zander et al., 2011; Bringmann and Krämer, 2008; Lamberg et al., 2004]. 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., 2011, pp. 13 – 14] and [Marrero Perez and Kaiser, 2009, pp. 2] discuss of how each testing setup adds detail to the testing representativeness, [Bringmann and Krämer, 2008, pp. 3] discusses the increasing level of integration of the different testing setups, [Bringmann and Krämer, 2006] and [Peleska, 2002] 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 the 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] and consistent injection of different types of faults (Section 4),

(iii) comparison and discussion of said setups in terms of their ability to detect different types of software faults (Section 5).

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

[1] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

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 [Desborough and Miller, 2002], 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 because they *all* implement an iterated loop of sensing, state estimation, control computation and actuation. Said 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 results of fault injections. Section 5 discusses our testing results and their generalisability and limitations. Section 6 and Section 7 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 [Lee and Seshia, 2016]. In practice, control software samples in real time a vector of measurements $y(t)$ from a physical process. 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 process. The state vector is linked to the vector $y(t)$ of measurements, creating a closed loop between the physical process 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* [Levine, 2009], 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 classical control system: vehicle cruise control. In this case, we want to regulate the vehicle longitudinal velocity, $v(t)$. Stability requires that the velocity eventually reaches a constant value, formally defined as $\lim_{t\to\infty} v(t) = \bar{v}$ where $\bar{v}$ is finite. Set-point tracking may be specified as: the absolute value of the difference between $\bar{v}$ and $r(t)$ is not greater than $2\,km/h$, $|\bar{v} - r(t)| \leq 2\,km/h$. The settling time requirement may impose a maximum of $10\,s$ to bring the vehicle velocity from $50\,km/h$ to $70\,km/h$: given $v(t_0) = 50\,km/h$ and $r(t_x) = 70\,km/h$ for $t_x \in (t_0, t]$, then $v(t) \in [68\,km/h, 72\,km/h]\, \forall t \geq t_0 + 10\,s$.

The next step in the development is the synthesis of a physical process model – usually a set of nonlinear differential equations. For the cruise control, these equations describe the longitudinal dynamics of the vehicle. Models are derived using first principles (from physics), data-driven methods, or a combination of the two [Åstrom and Murray, 2008].

Given the requirements and the model, the control engineer, together with application-domain-specific engineers, chooses which quantities must be measured and actuated—i.e. the sensors and actuators that have to be installed on the physical process. This defines the measurements and actuation signals available to the control algorithm. In the cruise control example, possible choices are: encoders mounted on the axes of the wheels to measure the car speed, a hydraulic pump to actuate the brakes of the car, and a servo motor to open or close the engine throttle.

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 [Levine, 2009]. 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 [Lamb, 2013; Murugesan et al., 2015]. Continuing with the cruise control example, the low-level equation-based controller is responsible for using the encoders measurements to actuate the throttle or brakes. The high-level discrete-state controller instead handles the control engagement and disengagement and other discrete inputs, e.g. signals from a collision detection system. 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 process. In our example, the cruise control algorithm is translated into code, compiled and flashed onto an electronic control unit on the vehicle.

The correctness of control software 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**) us-

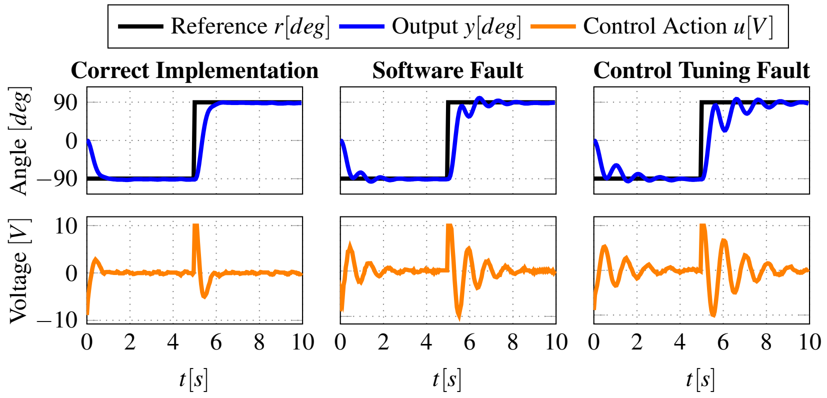


Figure 1. Step responses of the DC motor of the motivating example. The plots show how the control system makes the angular position of the motor (the blue line) follow the reference (the black line). The orange line shows the voltage actuated to the motor. The plot on the left-hand side shows the system performance with the correct implementaiton 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 wrong tuning of the control algorithm.

ing modelling assumptions that are not consistent with the physical process 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 process.

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* [Nguyen et al., 2018; Balasubramaniam et al., 2020]. 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 [Åstrom and Murray, 2008]. 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.

Motivating Example. We consider the control of a DC motor where the objective is to move the rotating motor to reach a desired angle.[2] 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[3] 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.

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 an incorrect tuning of the control algorithm: more specifically, a parameter is altered. This emulates a mistake of the *control engineer* when designing the control algorithm. 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. 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 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 control design fault. An *extra test on the sole model* of the control algorithm (hence directly using the differential equations designed by the control engineer) would abstract the software fault but still include the control tuning fault. This would allow to distinguish the responsibility of the faulty behaviour. If this extra test fulfils the requirements (i.e. doesn't show the oscillations of the rightmost plot), then the fault is in the software implementation: this conclusion can be made leveraging the fact that the software implementation is *abstracted* in the new setup that includes the model of the controller. Differently, if the extra test also fails, then

[2] Such systems are also commonly known as servo systems.

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

the fault is in the control design: this conclusion can be made since the control algorithm is *not abstracted* in either of the setups (and assuming that the model is implemented correctly).

This example shows two things: first, it shows how the step response can be used to verify the main control requirements. Secondly, 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 [Lamberg et al., 2004; Bringmann and Krämer, 2008].

3. Testing Abstractions

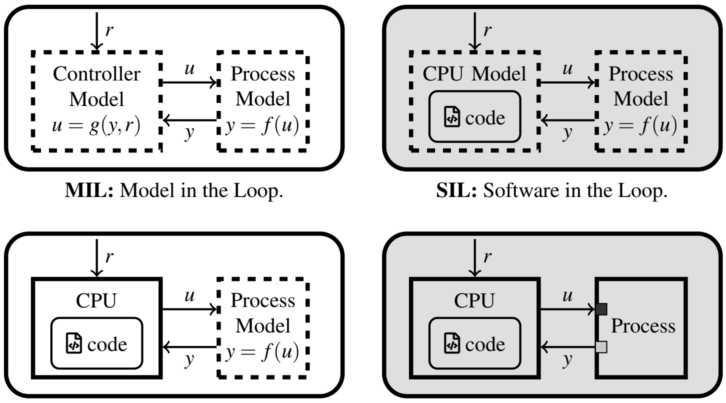
Section 2 showed the multi disciplinary nature of control systems. As a consequence, *system-level testing* is of fundamental importance. Like shown in our motivating example, it allows the engineers to establish the different responsibilities during the development process. Accordingly, it is one of the main activities software engineers perform in this context [Garca et al., 2020].

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 *process* block to form the closed loop. The system has three main components: (**i**) the physical process, (**ii**) the software implementing the control algorithm, and (**iii**) the hardware executing the software. The interaction between the controller and the physical process happens through actuators and sensors. The controller can also receive inputs from other software components or from human operators.

In the cruise control example, the hardware is the control unit (usually a microcontroller) mounted on the vehicle 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. commands from the steering wheel buttons to increase or decrease the speed).

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 [Zander et al., 2011]. 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,

Notation: r (reference commands), u (actuation signals), y (measured signals), ■ (actuators), □ (sensors).



MIL: Model in the Loop. **SIL:** Software in the Loop.

HIL: Hardware in the Loop. **PIL:** Process in the Loop.

Figure 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).

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 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 process. On top of these, in this paper we consider leveraging a hardware emulator.[4] 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

[4] 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.

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 Process in the Loop (PIL). The setup comprises controller code (C), hardware (H) and process (P); 🖩 indicates that a component is simulated, 🔌 that its real instance is used.

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

testing (so called model-testing [Briand et al., 2016]). For control design, the control engineer develops an executable model of the physical process, 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 [Levine, 2009], they can be implemented using common simulation software, like MATLAB[5] or Modelica.[6] 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 [Whalen et al., 2014].

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 process-related assumptions, and

(**ii**) controller-related assumptions.

Examples of assumptions on the process are: neglected dynamics (like tyre dynamics in the vehicle model for cruise control), 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.

[5] `https://www.mathworks.com/products/matlab.html`
[6] `https://modelica.org`

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. In the cruise control example, a controller handling the distance from the vehicle ahead and a controller keeping the desired speed might be tested separately. 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 process are still abstracted. The physical process is implemented using models that are similar to (or the same as) the ones used for MIL testing.[7] 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[8] and Renode,[9] 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 [Asadollah et al., 2018].

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 process 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:

[7] Here, models might need refinement. In the cruise control example, during the control design process, the engineer may assume direct control over the vehicle 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 connected to a servo, that moves the throttle valve of the engine.

[8] `http://www.gem5.org`

[9] `https://renode.io`

(**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 [Lohar et al., 2021]),

(**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).

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 of the car in the cruise control example – and the model of the physical process 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 process. 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 cruise control, speed 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 [Lee and Seshia, 2016] 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 process. 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 process.

3.4 Process in the Loop (PIL)

In the PIL setup the physical process 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 process and prototypes might be used in place of production models: such solutions are highly application-dependant and therefore excluded from this discussion. Hence, PIL testing effectiveness mostly depends on the testing strategy. However, in this work we focus on the design of the setups rather than of the testing strategy.

4. Experimental case study

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

We developed the setups with the objective of allowing consistent injection of the software faults in each of them. We did so by allowing only minimal modifications 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 use this procedure to expose the different fault-revealing capabilities of the setups.

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 [Carlos et al., 2020], swarm robotics [Araki et al., 2017;

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

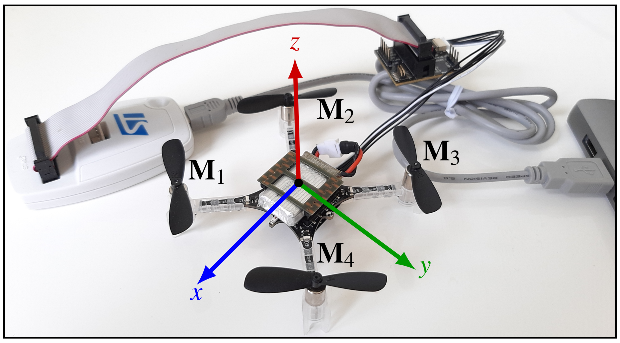


Figure 3. Crazyflie 2.1 with the STM debugger link.

Laclau et al., 2021; Mitchell et al., 2016], distributed [Wang et al., 2020] and robust control [Müller and D'Andrea, 2014]. Second, both the Crazyflie software[11] and hardware[12] 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 PIL setups. In particular, using the open source hardware specification, we can build the hardware emulator for the SIL testing. 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.

This section is organised as follows. In Subsection 4.1 we provide the relevant background information on the Crazyflie.[13] In Subsection 4.2 we describe our implementation of the testing setups. Finally, in Subsection 4.3 we both run tests with the nominal software and inject faults in the software. We report on the execution of the tests in each setup. A repository accompanies the submission,[14] providing the code we developed for the testing setups, documentation to reproduce the tests, but also pre-recorded flight data and detailed plots for each of the tests.

4.1 Crazyflie Quadcopter

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

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

[12] `https://github.com/bitcraze/hardware`

[13] 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`.

[14] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

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

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 [Smith et al., n.d.] 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.

When flying with optical flow data, the state estimator is implemented as an Extended Kalman Filter (**EKF**) [Mueller et al., 2015; Mueller et al., 2016], while the feedback controller is a set of cascaded PID controllers [Åström and Hägglund, 2006]. The setup with state-estimator and feedback controller is standard in control theory and found in most control systems.[20] The control design process provides equations used to model the quadcopter and equations to describe the estimator and the controller [Förster, 2015; Greiff, 2017]. Such equations can be found in our technical report.[21]

[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/dummy-testing-abstractions/cps-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.

[20] We describe the Crazyflie EKF and PID in a technical report in the repository `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`.

[21] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions/blob/main/Technical_Report.pdf`

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 process, so that the tests expose only differences in the control algorithm executions and the associated testing abstractions.[22] The testing setups of SIL, HIL and PIL require changes in the Crazyflie software, for which we provide patch files and application instructions.[23]

MIL. We implemented in Python a physical model to describe the Crazyflie and its controller [Greiff, 2017]. We use the SciPy module[24] 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[25] 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.[26] Bitcraze[27] maintains its own fork of Renode[28] and of the Renode-Infrastructure[29] which contains the emulators of the 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[30] interface. 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

[22] 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.

[23] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

[24] `https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html`

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

[26] `https://renode.io`

[27] `https://www.bitcraze.io/`

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

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

[30] `http://openocd.org/`

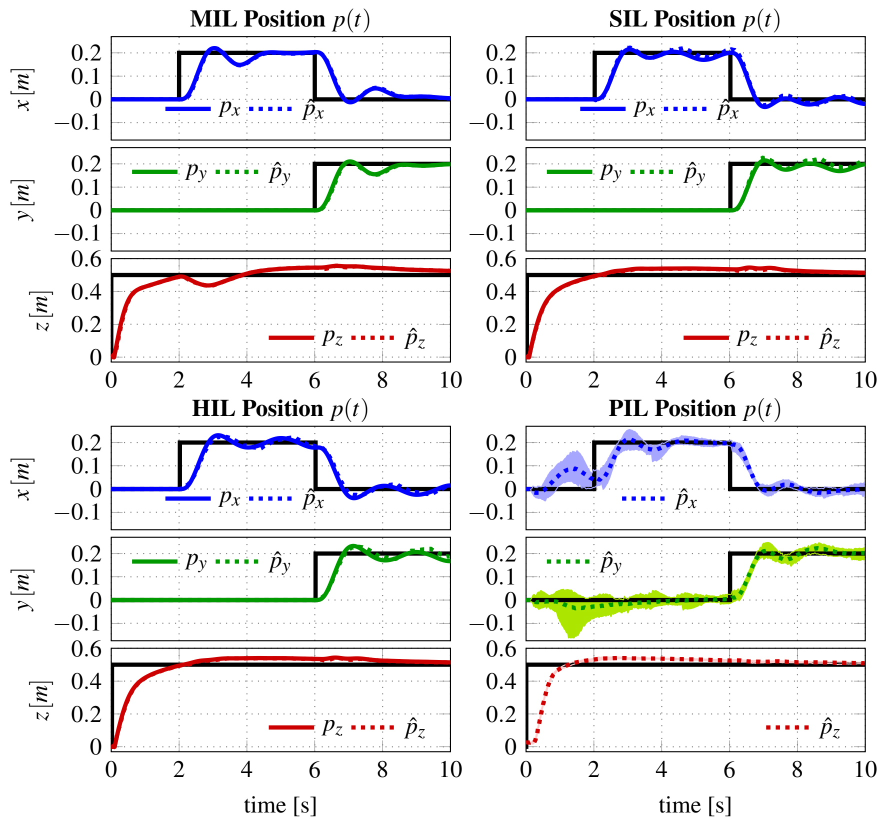


Figure 4. Nominal flight tests in the MIL, SIL, HIL, and PIL setups. For each axis x, y and z, the solid coloured lines show the drone's true position (when available). The black lines show the step references. The dashed lines show the estimated position. For the 30 repeated PIL flights, at each time 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.

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 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 process 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, to enable low-level access to the hardware, we used the debugger link ST-LINK/V2,[31] also depicted in Figure 3. We used OpenOCD[32] to interface with the debugger, and communicate with the CPU. OpenOCD exposes a Telnet port through which it is possible to read and write to specific memory addresses, or insert breakpoints. 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`,

[31] `https://www.st.com/en/development-tools/st-link-v2.html`
[32] `http://openocd.org/`

(**viii**) two assert statements in `uart_syslink.c` are skippeed.[33] 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 process 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.

PIL. Finally, our PIL testing setup consists of running the Crazyflie with its nominal software. We use the Micro SD card deck to log flight data.[34] 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 PIL 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 PIL with the nominal software to assess the repeatability of the PIL experiments.

4.3 Experimental Results

Here we describe our experimental results, using the MIL, SIL, HIL, and PIL setups. Initially, we discuss MIL tests results. We then introduce the implementation of the control software, first in its nominal state (as released by Bitcraze) and then after injecting different faults. We aim at *exposing the differences in the capability to uncover faults* that the testing setups offer. In Section 5, we comment our tests and discuss the research questions in light of our results.

Nominal Software. 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

[33] 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.

[34] `https://store.bitcraze.io/products/sd-card-deck`

(from $t = 0$ to $t = 2$), followed by a 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 [Åstrom and Murray, 2008]. 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) [Timperley et al., 2018]. Apparently, exhaustive testing of the controller implementation requires more tests, and test case generation for CPSs is an active research topic [Zander et al., 2011]. 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.[35] For each plot, the figure shows both the actual position from the simulated physics (coloured solid lines) and the drone's estimation (coloured dotted lines). The plots also include the reference position r (dark solid lines). This test shows that the model of the controller is able control the model of the process. Guarantees on the behaviour of the actual control system are however subject to the validity of both process and controller models, and on the implementation details [Alshahwan et al., 2019]—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 process in the PIL setup. In PIL, 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 PIL 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 PIL 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.

While the general behaviour is consistent across the setups, few differences arise. In the SIL, HIL and PIL 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

[35] More comprehensive plots for all the nominal and faulty test scenarios can be found at: `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

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 PIL setup. Such phenomenon is chaotic and difficult to model hence 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.

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[36] 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 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 [Steinbauer, 2013; Wienke et al., 2016] 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.

In [Wienke et al., 2016], the authors 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).

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

Table 2. List of injected faults and corresponding test results. The fault names correspond to the patch files and flight plots in the repository. For each fault, we report the corresponding categories covered among the types of faults in robotics faults highlighted in previous literature [Steinbauer, 2013; Wienke et al., 2016]. For each fault and setup, we report in the last three columns whether the test flight was impaired ✗ or not ✈. We note naming discrepancies between the two works: e.g. a missed deadline is considered a threading fault in [Wienke et al., 2016] and algorithmic in [Steinbauer, 2013]. 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., 2016]	Category from [Steinbauer, 2013]	SIL	HIL	PIL
`voltageCompCast`	—	batteries/low-level drivers	✗	✗	✗
`initialPos`	configuration	algorithm: configuration	✗	✗	✗
`flowGyroData`	threading	sensors: communication	✗	✗	✗
`motorRatioDef`	—	motors driver/low-level drivers	✗	✗	✈
`simUpdate`	algorithms & logic	algorithm: wrong estimation	✈	✈	✈
`byteSwap`	—	sensors: connectors/config.	✗	✈	✗
`gyroAxesSwap`	—	sensors: connectors/config.	✗	✈	✗
`timingKalman`	threading	algorithms: missed deadlines	✗	✈	✈
`flowDeckdtTiming`	threading	software: computer vision	✗	✈	✈
`slowTick`	configuration	platform: controller board	✈	✈	✗

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.

In [Steinbauer, 2013], the authors surveyed the participants to the RoboCup[37] 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 [Wienke et al., 2016; Steinbauer, 2013]. For each fault, we provide a patch file that injects it in the software.[38] After injecting a fault, we perform tests in the SIL, HIL and PIL 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.

[37] `https://www.robocup.org/`

[38] The repository (`https://github.com/dummy-testing-abstractions/cps-testing-abstractions`) contains information about the specific version of the software that we used, together with detailed instructions on how to retrieve the correct version and inject the faults.

Faults Description and Setup Detection Analysis. 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 by comparing to the nominal behaviour observed in Figure 4. Complete flight data and pre-generated plots are available at,[39] respectively inside the `flightdata` and `pdf` subfolders for each setup. We also report 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 said plots, we show only the position along the x-axis, both the ground truth p_x and its estimated value $\hat{p}_x$, as it suffices our discussion.

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 exposition that we observe are due to our specific implementation choices of the setups or are associated to more general properties. 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.

The `voltageCompCast` and `initialPos` faults are taken from the Bitcraze repository.[40] 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 $1.5m$. 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 drone flight results in 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 PIL.

[39] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

[40] `https://github.com/bitcraze/crazyflie-firmware/issues/766`, `https://github.com/bitcraze/crazyflie-firmware/issues/760`

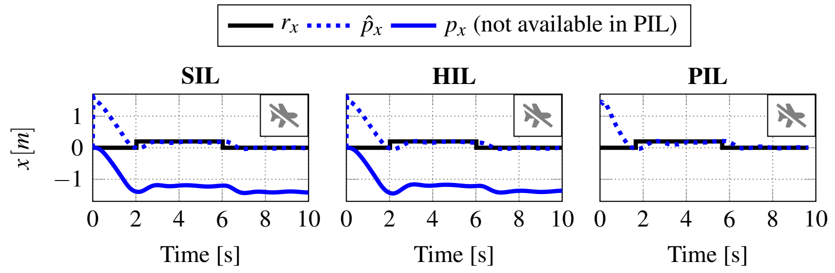


Figure 5. SIL, HIL and PIL plots of x-axis position with the `initialPos` fault. This fault affects the initialization of the state estimator and appears equivalently in each setup. These tests show that faults in the functional aspects of the software appear equivalently across the setups.

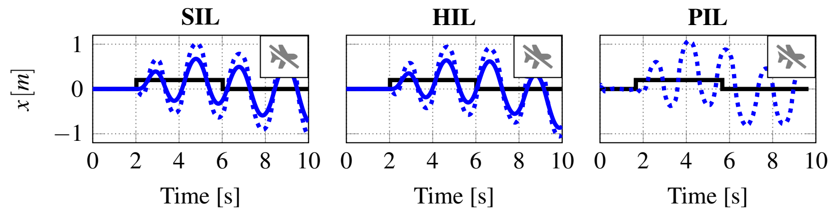


Figure 6. SIL, HIL and PIL plots of x-axis position with the `flowGyroData` fault. This fault affects the fusion of the inertial and visual odometry data and appears equivalently in each setup, more specifically the optical flow data with the attitude rate. These tests show that faults in the functional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

This means that, the abstraction introduced by the use of the physics 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 PIL 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.[41] 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

[41] 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.

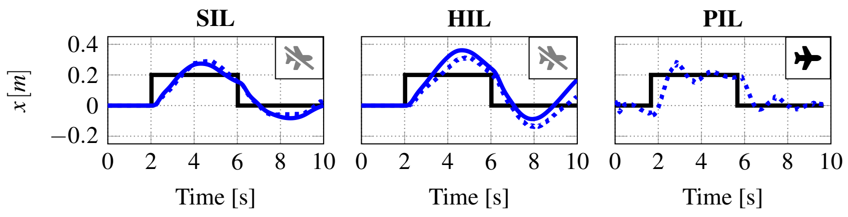


Figure 7. SIL, HIL and PIL plots of x-axis position with the `motorRatioDef` fault. This fault is caused by different software components assuming different definitions of the same quantity: in this case the motor actuation value. As it affects the interaction with the low-level driver of the motors,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

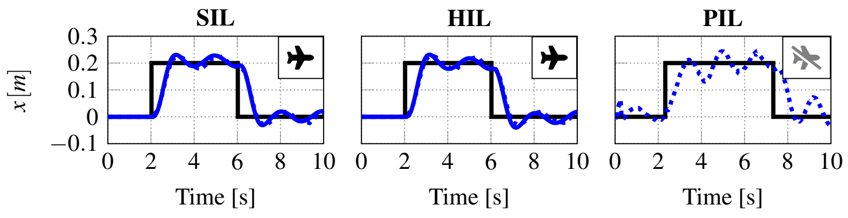


Figure 8. SIL, HIL and PIL plots of x-axis position with the `slowTick` fault. This fault affects in the timing in the real-time operative system. As the timing of the code execution is abstracted in different ways it appears differently in the setups. Specifically it shows a limitation in capturing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

direction in our setups. In the PIL 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[42]) 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.

[42] `https://github.com/dummy-testing-abstractions/cps-testing-abstractions`

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 PIL, 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. 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 PIL. 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 timing of the software. Achieving this is however a challenging task and high-fidelity emulation of time execution of software is an open research problem. In `slowTick`, a hardware clock malfunction is simulated by setting the RTOS software tick to $800Hz$ instead of $1000Hz$. 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 PIL. 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 PIL 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.

5. 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 exposition of functional faults with respect to the PIL 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 [Sotiropoulos et al., 2017]).

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 level. This is due to the abstraction in HIL of the low-level interaction with the IMU. For our system, the SIL setup has a better code coverage than HIL. Generalising this consideration, we can say that *when the code is executed in an emulated environment, the low-level interaction with the hardware can be fully emulated. Conversely, when the target hardware is used, the code needs to be manipulated to inject the inputs and read the outputs*. This limits the testing capabilities for some software components in HIL. More specifically, faults in the low level software – which has been reported to be prone to faults in robotic systems [Steinbauer, 2013; Wienke et al., 2016] – can be hidden. As mentioned in Section 3.3, to improve the low-level code coverage of the HIL abstraction level setup, 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 implementation effort to emulate sensors.

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 PIL, 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 PIL 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 PIL (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 PIL, 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 [Taylor et al., 2014a; Taylor et al., 2014b].

The remaining timing-related fault, `slowTick`, shows a limitation that is common to both SIL and HIL. In both setups, the simulation of the physic 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 PIL setup, it is possible to detect this fault, since there is no modelled physics, the physical part is 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. PIL 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 PIL 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 PIL 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 [Luo et al., 2014]; 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,
- SIL provides better low-level code coverage with respect to HIL,
- HIL provides 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 improved coverage comes from 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.[43]

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)

[43] 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.

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 [Garca et al., 2020]. 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 [Åstrom and Murray, 2008]. Said robustness varies depending on the process 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*.

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 formulating and meeting 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 process, 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. The development of the software is also different, as regulations constrain the verification and validation process [Moy et al., 2013]. Finally, physical processes 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. However, 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 [Steinbauer, 2013; Wienke et al., 2016]. 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 process 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.

6. Related Work

Recent research highlighted interesting research directions at the intersection of control and software engineering [Balasubramaniam et al., 2020; Bradley and Bagheri, 2020]. In the control literature, Zimmer et al. [Zimmer et al., 2015] 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 [Zander et al., 2011]; another review can be found in [Garousi et al., 2018; Banerjee et al., 2016].

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 [Peleska, 2002; White, 2001] and automotive [Bringmann and Krämer, 2008; Bringmann and Krämer, 2006].

The concept of testing abstractions is discussed in [Zander et al., 2011], and the testing setups discussed in this paper appear in different work, with slightly varying but overall consistent definitions [Zander et al., 2011; Bringmann and Krämer, 2008; Lamberg et al., 2004]. The MIL setup has been extensively leveraged in the literature of model based testing [Briand et al., 2016]. The research has been focused on: verification of requirements [Nejati et al., 2019], generation of test traces [Hänsel et al., 2011], or the use of models for the automatic generation of test cases with search algorithms [Matinnejad et al., 2014; Matinnejad et al., 2017; Marculescu et al., 2015], classification trees [Lamberg et al., 2004], system-identification based refinements [Menghi et al., 2019b], or genetic algorithms [Aleti and Grunske, 2015]. In robotics, [Silano et al., 2018] showcased the usefulness of SIL for the design of quadcopter controllers. Keranen et al. [Keränen and Räty, 2013] perform a validation of model based approaches in the context of HIL testing, and Hansen et al. [Hansen et al., 2017] do the same in the context of SIL testing. Meedeniya et al. [Meedeniya et al., 2011] propose an optimisation for reliable deployment of control software. Ore et al. [Ore et al., 2018] 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, this is the first study on the comparison of the different testing setups and the associated abstractions.

7. Conclusion

In this paper, we provided a comparison of the characteristics with respect to fault finding of the model-in-the-loop, software-in-the-loop, hardware-in-the-loop and process-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 abstractions. 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; however, there are not major differences in terms of exposition 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 the engineers at Bitcraze, and in particular Marcus Eliasson for the support with the Crazyflie platform and the insights on the development of the testing setups.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH). The responsibility for the content remains with the authors.

References

Afzal, A., C. L. Goues, M. 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)*, pp. 96–107.

Aleti, A. and L. Grunske (2015). "Test data generation with a kalman filter-based adaptive genetic algorithm". *Journal of Systems and Software* **103**, pp. 343–352. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.11.035. URL: https://www.sciencedirect.com/science/article/pii/S0164121214002660.

Alshahwan, N., A. Ciancone, M. Harman, Y. Jia, K. Mao, A. Marginean, A. Mols, H. Peleg, F. Sarro, and I. 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, Beijing, China, pp. 1–3. ISBN: 9781450362245. DOI: 10.1145/3293882.3338991. URL: https://doi.org/10.1145/3293882.3338991.

Araki, B., J. Strang, S. Pohorecky, C. Qiu, T. Naegeli, and D. 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, pp. 5575–5582. DOI: 10.1109/ICRA.2017.7989657. URL: https://doi.org/10.1109/ICRA.2017.7989657.

Asadollah, S. A., D. Sundmark, S. Eldh, and H. 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)*, pp. 172–179. DOI: 10.1109/ISPDC2018.2018.00032.

Åstrom, K. J. and R. M. Murray (2008). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA. ISBN: 0691135762.

Åström, K. J. and B. Wittenmark (2013). *Computer-controlled systems: theory and design*. Courier Corporation.

Åström, K. and T. Hägglund (2006). *Advanced PID Control*. English. ISA - The Instrumentation, Systems and Automation Society. ISBN: 978-1-55617-942-6.

Bach, J., J. Langner, S. Otten, E. Sax, and M. 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)*, pp. 203–210. DOI: 10.1109/ICE.2017.8279890.

Balasubramaniam, B., H. Bagheri, S. Elbaum, and J. Bradley (2020). "Investigating controller evolution and divergence through mining and mutation*". In: *2020*

ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS), pp. 151–161. DOI: 10.1109/ICCPS48487.2020.00022.

Banerjee, A., S. Chattopadhyay, and A. Roychoudhury (2016). “Chapter three - on testing embedded software”. In: Memon, A. (Ed.). Vol. 101. Advances in Computers. Elsevier, pp. 121–153. DOI: https://doi.org/10.1016/bs.adcom.2015.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0065245815000662.

Bertolino, A., P. Braione, G. D. Angelis, L. Gazzola, F. Kifetew, L. Mariani, M. Orrù, M. Pezzè, R. Pietrantuono, S. Russo, and P. Tonella (2021). “A survey of field-based testing techniques”. *ACM Comput. Surv.* **54**:5. ISSN: 0360-0300. DOI: 10.1145/3447240. URL: https://doi.org/10.1145/3447240.

Bradley, J. M. and H. Bagheri (2020). “Control software: research directions in the intersection of control theory and software engineering”. In: *AIAA Scitech 2020 Forum*. DOI: 10.2514/6.2020-2102. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2020-2102. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2020-2102.

Briand, L., S. Nejati, M. Sabetzadeh, and D. 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, Austin, Texas, pp. 789–792. ISBN: 9781450342056. DOI: 10.1145/2889160.2889212. URL: https://doi.org/10.1145/2889160.2889212.

Bringmann, E. and A. 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, Shanghai, China, pp. 13–20. ISBN: 1595934022. DOI: 10.1145/1138474.1138479. URL: https://doi.org/10.1145/1138474.1138479.

Bringmann, E. and A. Krämer (2008). “Model-based testing of automotive systems”. In: *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 485–493. DOI: 10.1109/ICST.2008.45.

Broy, M., I. H. Kruger, A. Pretschner, and C. Salzmann (2007). “Engineering automotive software”. *Proceedings of the IEEE* **95**:2, pp. 356–373. DOI: 10.1109/JPROC.2006.888386.

Carlos, B. B., T. Sartor, A. Zanelli, G. Frison, W. Burgard, M. Diehl, and G. 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, pp. 982–989. DOI: 10.1109/ICARCV50220.2020.9305513. URL: https://doi.org/10.1109/ICARCV50220.2020.9305513.

Desborough, L. and R. Miller (2002). “Increasing customer value of industrial control performance monitoring honeywell s experience”. In:

Förster, J. (2015). *System identification of the crazyflie 2.0 nano quadrocopter*.

Garca, S., D. Strüber, D. Brugali, T. Berger, and P. 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, Virtual Event, USA, pp. 593–604. ISBN: 9781450370431. DOI: `10.1145/3368089.3409743`. URL: `https://doi.org/10.1145/3368089.3409743`.

Garousi, V., M. Felderer, Ç. M. Karapçak, and U. Ylmaz (2018). “Testing embedded software: a survey of the literature”. *Information and Software Technology* **104**, pp. 14–45. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2018.06.016`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584918301265`.

Grässler, I., E. Bodden, J. Pottebaum, J. Geismann, and D. Roesmann (2020). “Security-oriented fault-tolerance in systems engineering: a conceptual threat modelling approach for cyber-physical production systems”. In: Bartoszewicz, A. et al. (Eds.). *Advanced, Contemporary Control*. Springer International Publishing, Cham, pp. 1458–1469.

Greiff, M. (2017). *Modelling and control of the crazyflie quadrotor for aggressive and autonomous flight by optical flow driven state estimation*. eng. Student Paper.

Hänsel, J., D. Rose, P. Herber, and S. 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*, pp. 170–179. DOI: `10.1109/ICST.2011.37`.

Hansen, N., N. Wiechowski, A. Kugler, S. Kowalewski, T. 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*.

Keränen, J. and T. Räty (2013). “Validation of model-based testing in hardware in the loop platform”. In: *2013 10th International Conference on Information Technology: New Generations*, pp. 331–336. DOI: `10.1109/ITNG.2013.53`.

Laclau, P., V. Tempez, F. Ruffier, E. Natalizio, and J. Mouret (2021). “Signal-based self-organization of a chain of uavs for subterranean exploration”. *Frontiers Robotics AI* **8**, p. 614206. DOI: `10.3389/frobt.2021.614206`. URL: `https://doi.org/10.3389/frobt.2021.614206`.

Lamb, F. (2013). *Industrial Automation: Hands On*. McGraw-Hill Professional, US. ISBN: 0071816453. DOI: `10.1036/9780071816472`.

Lamberg, K., M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey (2004). "Model-based testing of embedded automotive software using mtest". In: *SAE 2004 World Congress and Exhibition*. SAE International. DOI: `https://doi.org/10.4271/2004-01-1593`. URL: `https://doi.org/10.4271/2004-01-1593`.

Lee, E. A. and S. A. Seshia (2016). *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. The MIT Press. ISBN: 0262533812.

Levine, W. S. (2009). *The Control Systems Handbook*. 2nd. CRC Press, Inc., USA. ISBN: 1420073648.

Lohar, D., C. Jeangoudoux, J. Sobel, E. 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**, pp. 43–63.

Luo, Q., F. Hariri, L. Eloussi, and D. 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, Hong Kong, China, pp. 643–653. ISBN: 9781450330565. DOI: `10.1145/2635868.2635920`. URL: `https://doi.org/10.1145/2635868.2635920`.

Maia, P. H., L. Vieira, M. Chagas, Y. Yu, A. Zisman, and B. 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)*, pp. 107–113. DOI: `10.1109/SEAMS.2019.00022`.

Marculescu, B., R. Feldt, R. Torkar, and S. Poulding (2015). "An initial industrial evaluation of interactive search-based testing for embedded software". *Applied Soft Computing* **29** (0), pp. 26–39. DOI: `http://dx.doi.org/10.1016/j.asoc.2014.12.025`. URL: `http://www.sciencedirect.com/science/article/pii/S1568494614006693`.

Marrero Perez, A. and S. Kaiser (2009). "Integrating test levels for embedded systems". In: *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pp. 184–193. DOI: `10.1109/TAICPART.2009.22`.

Matinnejad, R., S. Nejati, L. Briand, and T. 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, Vasteras, Sweden, pp. 163–174. ISBN: 9781450330138. DOI: `10.1145/2642937.2642978`. URL: `https://doi.org/10.1145/2642937.2642978`.

Matinnejad, R., S. Nejati, and L. 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, Paderborn, Germany, pp. 938–943. ISBN: 9781450351058. DOI: `10.1145/3106237.3117770`. URL: `https://doi.org/10.1145/3106237.3117770`.

Maxwell, J. C. (2011). “On governors”. In: Niven, W. D. (Ed.). *The Scientific Papers of James Clerk Maxwell*. Vol. 2. Cambridge Library Collection - Physical Sciences. Cambridge University Press, pp. 105–120. DOI: `10.1017/CBO9780511710377.009`.

Meedeniya, I., B. Buhnova, A. Aleti, and L. Grunske (2011). “Reliability-driven deployment optimization for embedded systems”. *Journal of Systems and Software* **84**:5, pp. 835–846. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2011.01.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121211000069`.

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

Menghi, C., S. Nejati, L. C. Briand, and Y. I. Parache (2019b). “Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification”. *CoRR* **abs/1910.02837**. arXiv: `1910.02837`. URL: `http://arxiv.org/abs/1910.02837`.

Mitchell, D., E. A. Cappo, and N. 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, pp. 4810–4815. DOI: `10.1109/IROS.2016.7759706`. URL: `https://doi.org/10.1109/IROS.2016.7759706`.

Molzahn, D. K., F. Dörfler, H. Sandberg, S. H. Low, S. Chakrabarti, R. Baldick, and J. Lavaei (2017). “A survey of distributed optimization and control algorithms for electric power systems”. *IEEE Transactions on Smart Grid* **8**:6, pp. 2941–2962. DOI: `10.1109/TSG.2017.2720471`.

Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate (2013). “Testing or formal verification: do-178c alternatives and industrial experience”. *IEEE Software* **30**:3, pp. 50–57. DOI: `10.1109/MS.2013.43`.

Mueller, M. W., M. Hamer, and R. DAndrea (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)*, pp. 1730–1736. DOI: `10.1109/ICRA.2015.7139421`.

Mueller, M. W., M. Hehn, and R. DAndrea (2016). “Covariance correction step for kalman filtering with an attitude”. *Journal of Guidance, Control, and Dynamics*, pp. 1–7.

Müller, H. A. (2017). “The rise of intelligent cyber-physical systems”. *Computer* **50**:12, pp. 7–9. DOI: `10.1109/MC.2017.4451221`.

Müller, M. W. and R. 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, pp. 45–52. DOI: 10.1109/ICRA.2014.6906588. URL: https://doi.org/10.1109/ICRA.2014.6906588.

Murugesan, A., S. Rayadurgam, M. W. Whalen, and M. P. E. Heimdahl (2015). "Design considerations for modeling modes in cyberphysical systems". *IEEE Design Test* **32**:5, pp. 66–73. DOI: 10.1109/MDAT.2015.2462112.

Nejati, S., K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. 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, Tallinn, Estonia, pp. 1015–1025. ISBN: 9781450355728. DOI: 10.1145/3338906.3340444. URL: https://doi.org/10.1145/3338906.3340444.

Nguyen, L. V., K. A. Hoque, S. Bak, S. Drager, and T. T. Johnson (2018). "Cyber-physical specification mismatches". *ACM Trans. Cyber-Phys. Syst.* **2**:4. ISSN: 2378-962X. DOI: 10.1145/3170500. URL: https://doi.org/10.1145/3170500.

Nilsson, P., O. Hussien, A. Balkan, Y. Chen, A. D. Ames, J. W. Grizzle, N. Ozay, H. Peng, and P. Tabuada (2016). "Correct-by-construction adaptive cruise control: two approaches". *IEEE Transactions on Control Systems Technology* **24**:4, pp. 1294–1307. DOI: 10.1109/TCST.2015.2501351.

Ore, J.-P., C. Detweiler, and S. 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, Gothenburg, Sweden, pp. 40–43. ISBN: 9781450357609. DOI: 10.1145/3196558.3196566. URL: https://doi.org/10.1145/3196558.3196566.

Peleska, J. (2002). "Hardware/software integration testing for the new airbus aircraft families." *http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html*. DOI: 10.1007/978-0-387-35497-2_24.

Silano, G., E. Aucone, and L. 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)*, pp. 1–6. DOI: 10.1109/MED.2018.8442759.

Smith, N. T., J. T. Heineck, and E. T. Schairer (n.d.). "Optical flow for flight and wind tunnel background oriented schlieren imaging". In: *55th AIAA Aerospace Sciences Meeting*. DOI: 10.2514/6.2017-0472. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2017-0472. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2017-0472.

Sotiropoulos, T., H. Waeselynck, J. Guiochet, and F. 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)*, pp. 150–159. DOI: `10.1109/QRS.2017.25`.

Steinbauer, G. (2013). “A survey about faults of robots used in robocup”. In: Chen, X. et al. (Eds.). *RoboCup 2012: Robot Soccer World Cup XVI*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 344–355. ISBN: 978-3-642-39250-4.

Taylor, J. R., E. M. Drumwright, and G. Parmer (2014a). “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, Bergamo, Italy, pp. 1–12. ISBN: 9783319118994. DOI: `10.1007/978-3-319-11900-7_1`. URL: `https://doi.org/10.1007/978-3-319-11900-7_1`.

Taylor, J. R., E. M. Drumwright, and G. Parmer (2014b). “Temporally consistent simulation of robots and their controllers”. In: vol. Volume 6: 10th International Conference on Multibody Systems, Nonlinear Dynamics, and Control. International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. DOI: `10.1115/DETC2014-35609`. URL: `https://doi.org/10.1115/DETC2014-35609`.

Timperley, C. S., 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)*, pp. 331–342. DOI: `10.1109/ICST.2018.00040`.

van der Knijff, R. (2014). “Control systems/scada forensics, what’s the difference?” *Digital Investigation* **11**:3. Special Issue: Embedded Forensics, pp. 160–174. ISSN: 1742-2876. DOI: `https://doi.org/10.1016/j.diin.2014.06.007`. URL: `https://www.sciencedirect.com/science/article/pii/S1742287614000814`.

Wang, G., W. Yang, N. Zhao, Y. Ji, Y. Shen, H. Xu, and P. 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, pp. 3234–3240. DOI: `10.1109/ICRA40945.2020.9196926`. URL: `https://doi.org/10.1109/ICRA40945.2020.9196926`.

Whalen, M. W., A. Murugesan, S. Rayadurgam, and M. 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, Hyderabad, India, pp. 19–24. ISBN: 9781450328494. DOI: `10.1145/2593770.2593776`. URL: `https://doi.org/10.1145/2593770.2593776`.

White, A. (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. DOI: `10.1109/AERO.2001.931302`.

Wienke, J., S. Meyer zu Borgsen, and S. Wrede (2016). “A data set for fault detection research on component-based robotic systems”. In: Alboul, L. et al. (Eds.). *Towards Autonomous Robotic Systems*. Springer International Publishing, Cham, pp. 339–350. ISBN: 978-3-319-40379-3.

Zander, J., I. Schieferdecker, and P. Mosterman (2011). *Model-Based Testing for Embedded Systems*. ISBN: 9781439818459.

Zheng, X., C. Julien, M. Kim, and S. Khurshid (2017). “Perceptions on the state of the art in verification and validation in cyber-physical systems”. *IEEE Systems Journal* **11**:4, pp. 2614–2627. DOI: `10.1109/JSYST.2015.2496293`.

Zimmer, M., J. Hedrick, and E. A. Lee (2015). “Ramifications of software implementation and deployment: a case study on yaw moment controller design”. *2015 American Control Conference (ACC)*, pp. 2014–2019.

Paper III

Testing of Control-Based Cyber-Physical Systems

Claudio Mandrioli, Seung Yeob Shin, Martina Maggio,
Domenico Bianculli, Lionel Briand

Abstract

Cyber-Physical Systems (CPS) are most times safety-critical and expected to perform in uncertain environments. Therefore the identification of which scenarios prevent the CPS from performing according to its requirements is of fundamental importance. However, the multidisciplinary nature of CPS can make is difficult to identify such scenarios. In this paper we discuss the testing of CPS that are developed with the use of control theory. In such systems, the software is developed partially by the control engineers and partially by the software engineers. When testing, it is important to account for this multidisciplinary development. We control engineers make different sets of design assumptions when contributing to the system development. However, such assumptions are not always satisfied in the implemented system. We then define the problem of stress testing control-based CPS as the generation and identification of test cases that invalidate said design assumptions. Among the listed sets of assumptions, we highlight the use of linearised models of the physics. With focus on the linear model assumptions, and on the base of control-theoretical background, we develop a qualitative characterisation of the input space of the control layer in CPS. We then propose a novel test case parametrisation for control-based CPS and use it together with the proposed characterisation to develop a testing approach aiming at falsifying the targeted assumptions. We evaluate our testing approach on six case of studies. Our results show that the proposed testing approach is effective at invalidating the linearity design assumption and that it can be used to highlight the components of the CPS that can limit the scenarios in which it can fulfil its requirements.

Under Submission to Transactions of Software Engineering and Methodology.

1. Introduction

Cyber-Physical Systems (CPS) are engineering artefacts characterised by the tight coupling of physical and software components [Lee, 2015]. This tight coupling is created by sensors and actuators that allow the software component to measure and affect the physical part of the system. Figure 1 graphically shows this interaction between the cyber component (green dashed box) and a physical component (red dashed box). The objective of this interaction is to obtain a desired behaviour of the physical component. Accordingly, the CPS requirements are generally defined over quantities that live in the physical part of the system, this is highlighted by the *CPS output* arrow in the figure. In the cyber component, we highlight the control layer. The control layer receives desired values for given physical quantities (the *references*) and uses sensors and actuators in order to enforce said values in the physical component. For example, a drone uses sensors (such as accelerometers and cameras) to estimate its position and actuators (such as propellers) to move. The desired behaviour is usually that the drone performs stable flight and reaches a desired position.

CPSs are often by nature safety-critical, and they are expected to operate in uncertain environments [Wu et al., 2017]. For example, drones and cars operate in environments where people are present and the external conditions are never fully known (e.g., the presence of wind and obstacles for the drone, and other vehicles and pedestrians on the road for the car). In such circumstances, it is of primary importance to identify the scenarios in which the CPS is no longer able to fulfil its requirements. To identify such scenarios, *stress testing* aims at executing a system under test (SUT) in conditions that are different from the ones expected during the system design [Priyadarshi Tripathy, 2008]. The execution and analysis of stress tests should provide the engineers with information on the design choices that limit the SUT capability of fulfilling its requirements in different scenarios.

However, CPS development is known to be multidisciplinary [Lee, 2015]. Accordingly, the definition of the operating conditions expected during the CPS design will depend on the design choices of different types of engineers (e.g., software engineers and control engineers), as well as on their combinations. For example, for a drone we want to identify what can limit its capability to avoid a moving obstacle. The limiting factors can be the hardware design (e.g., the sizing of the propellers), different software components (e.g. ,the path planning), or their interaction.

Intuitively, being composed of a physical and software part, CPS development involves both software engineers and engineers with specific knowledge of the physical part of the system (e.g., aerospace engineers for a drone or mechanical engineers for a car). However, besides these categories of engineers, many CPS applications involve also control engineers [He et al., 2019]. We call CPS that involve control engineers *control-based CPS*. In such applications, there are multiple levels of decision making, depending on the decision relevance and time-scale. For example, for a delivery drone, high-level decision making concerns the definition of

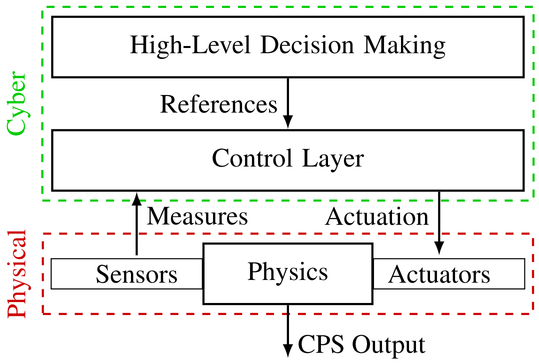


Figure 1. Structure of a CPS, where a cyber component (green dashed box) interacts with a physical component (red dashed box). The CPS requirements are generally defined over quantities that live in the physical part of the system, the *CPS output* arrow. In the cyber component, we highlight the control layer. The control layer receives desired values for given physical quantities (the *references*) and uses sensors and actuators in order to enforce said values in the physical component.

the sequences of picking up the object to be delivered and the trajectory planning. As shown in Figure 1, at the lowest level we find the *control-layer* that handles the real-time interaction with the physics. In our drone example, the control layer adjusts the propellers commands to react to wind gusts and maintain the desired position. In such systems, the role of control engineers is to design different *control algorithms* that are implemented as part of the control-layer. The control layer provides high-level components (like the “High-Level Decision Making” block in Figure 1) with an interface to control the physical component. Broadly speaking, it allows the high-level components to provide desired values (i.e., the *reference values*) for some specific physical quantities. It then uses readings from the sensors to decide actuation commands and force the actual physical quantities (the *CPS output*) to reach the desired values. For example, for a drone, the control layer allows the high-level components to set the desired drone position. It then uses the accelerometer and camera data to estimate the current drone position, and determines the propellers voltage commands to bring the drone to the target location.

When designing control algorithms, control engineers leverage control theory. To apply the theory, they make *design assumptions* and abstract certain aspects of the design problem. Such abstractions concern both the physical component as well as the software implementation of the control layer. For example, for a drone, they simplify the behaviour of the device, and neglect other functionalities of the software like flight mode changes. Although, from a strictly theoretical point of view, control theory can provide formal guarantees on the CPS performance (e.g., drone flight speed along a certain trajectory), in practice the design assumptions do not necessarily hold. We can then distinguish scenarios in which the design assumptions hold (and therefore also the formal guarantees are valid) from scenarios in which they are *falsified*. The scenarios in which the assumptions hold (e.g., values of the references, environment conditions, system state) define the *design scope* of the CPS. Conversely, other scenarios, where the software implementation of the control layer does not fulfil the design assumptions, are outside of the design scope of the algorithm and lead to the loos of the a priori guarantees. Said in other words, the design assumptions limit the control layer design scope, i.e., they reduce the number of scenarios in which the designed algorithm can provide a priori guarantees.

In this work, we use the falsification of the control design assumptions to define the problem of stress testing control-based CPSs software. We assessed our testing methodology by applying it two different case studies: the altitude control of a drone and the position control of a DC servo (a continuous current motor). Our results prove the capability of our approach and metrics of respectively generating and identifying test cases that invalidate the control design assumptions. Our test cases invalidate the design assumptions to different degrees and allow to observe the behaviour of the SUT at the bounds of its design scope. We showcase how this tests can be used to gain insight on the design choices that limit the ability of the SUTs to perform safely in different scenarios.

To summarise, this article makes the following contributions:

(i) We use control engineering domain knowledge to develop an input space qualitative characterisation for individual elements of the control layer real-valued inputs (Section 4.1).

(ii) We define different metrics to make quantitative the relevant aspects of the proposed characterisation (Section 4.2).

(iii) We propose a novel test case parameterisation for CPS control-layers that enables the use of the characterisation (Section 5.1). We use such parameterisation to develop a control-based CPS stress testing approach (Section 5.2).

The rest of the article is structured as follows. First, given the multidisciplinary nature of this problem, we present in Section 2 the relevant background on the control-based CPSs development and the control algorithms design process. We complement this with the discussion of the related work from the software engineering literature on the testing of control-based CPSs. Section 3 provides a control engineering perspective of CPS stress testing. In Section 4 we describe our characterisation of the control loop input space. We illustrate our testing approach in Section 5. Section 6 reports on our empirical evaluation. Section 7 concludes the article and outlines directions for future work.

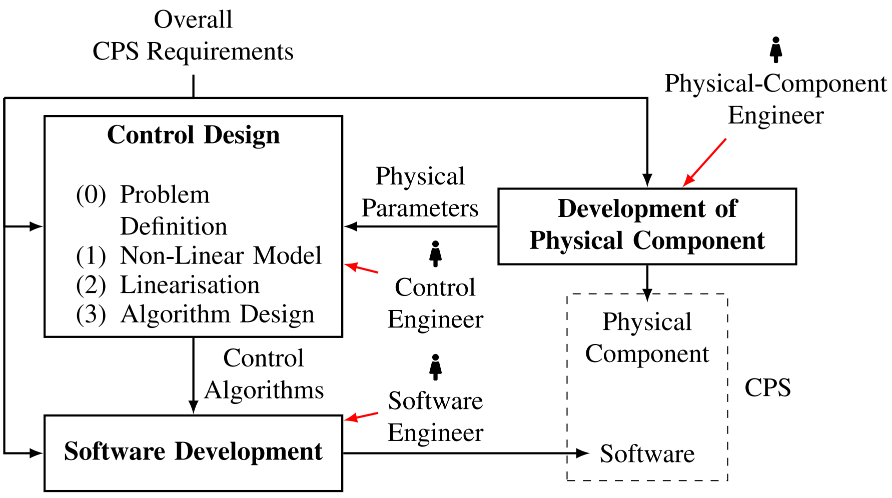


Figure 2. High-level description of the control system development; $\rightarrow$ represent development steps, while $\rightarrow$ represent data flow. The development flow is simplified (e.g. neglecting iterations) to focus on the role of control engineering.

2. Context and Background

This section provides the relevant background on the development of control-based CPS; we use the position control of a drone as running example to exemplify the different concepts. The section is divided in three parts. First, in Section 2.1 we present the engineering process for developing a control system, and define the key roles involved in this process. Then, in Section 2.2 we provide the relevant control engineering background needed to understand the remainder of this article. Finally, in Section 2.3 we present related work on the testing process for cyber-physical systems and control systems in particular.

2.1 Development of Control-Based CPS

Figure 2 provides a graphical overview of the typical development workflow of control-based CPS. The overview is simplified and focuses on the role that control engineers play in the development of the software handling the interaction between the physical and the cyber components.

The development of any engineering system starts with the definition of the system's requirements, denoted by *Overall CPS Requirements* in the figure. In the case of control-based CPSs, the requirements usually describe the desired behaviour of the *physical* part of the system. In our running example, the overall CPS requirements describe, at a high level, how the drone is supposed to move in space. A concrete example of such a requirement could be that the drone must plan and follow a prescribed trajectory, (at every point in time) with discrepancy not higher than a given threshold (e.g., two meters), and reach the desired position within a given time distance (e.g., three minutes).

These requirements are then made available to the engineers involved in the development of the CPS. We identify three types of engineers involved, each having different duties. In Figure 2, we use red arrows to highlight their role in the development.

(i) *Physical-Component Engineers* design the *physical part* of the system. These could be for example mechanical or aerospace engineers depending on whether the considered CPS is a car or a drone. In our running example, an aerospace engineer may size up the propellers and draw the mechanical structure of the drone's body.

(ii) *Control engineers* select and design the control algorithms. In the case of our running example, this means designing the algorithms in charge of estimating the drone current position based on the sensors readings, and computing the commands to be sent to the drone motors so that it can fly.

(iii) *Software engineers* are in charge of the cyber part of the CPS. They implement the control algorithms, and design the functions that are needed to integrate the different software components (e.g., the high-level decision making in Figure 1). In our running example, there might be an aggressive and faster control algorithm when the drone is flying outdoors, compared to a safer one when the drone is flying indoors in a constrained space.

In Figure 2, we use black arrows to highlight the data and information flow during the development workflow. For instance, in our running example, the physical-component engineer communicates the data related to the propeller thrust to the control engineer, who then uses it in the *control design process*, further described in Section 2.2. The result of the control design process is a set of algorithms that are provided to the software engineers, so that they can integrate them with the rest of the software necessary to fly the drone (e.g., the functions that perform initial checks, sensor data acquisition, communication with the motors).

Consequences of Multidisciplinarity on the Testing Process: The software development flow highlights the multidisciplinary nature of control-based CPSs. This multidisciplinarity observed during the development has an impact on the testing process. In particular, for control CPSs, it becomes very difficult (if not impossible) to distinguish the testing of the control algorithm (i.e., does the chosen control algorithm fulfil the overall CPS requirements?) from the testing of its software implementation. Ideally, we would like to be able to verify the control algorithm and the control implementation independently, in order to obtain a better separation of concerns in the development process.

However, this ideal separation of concerns is not achievable. Testing of CPS software (i.e., the cyber component of a CPS) has very limited effectiveness without an integrated setup that includes the physics behaviour. In fact, the overall CPS requirements are defined on the behaviour of the physical component. Hence, in order to evaluate the compliance with these requirements, the physical component needs to be included in the testing setup. For instance, to evaluate how the drone follows a prescribed trajectory, we need to either include a model (and hence a simulation) of how the drone flies in space, or conduct tests with the drone itself in a specific physical space. Only in this way, we can simulate or measure the output position of the drone, and use it to assess that the distance between the actual position and the planned trajectory satisfies the prescribed requirement.

The need for testing the interaction between the cyber and the physical components of a CPS is particularly relevant for control software (i.e., the software implementing the control layer). In fact, the control software handles the low-level interactions with actuators and sensors (i.e., processing the sensor readings and computing the actuation commands). To obtain a valid interaction, both the control algorithm and its implementation have to be correct. This makes it difficult, if not impossible, to separate the testing of these two aspects.

In the specific case of stress testing, this impossibility to distinguish the testing of the control algorithm from the testing of its software implementation implies that we need to account for the design scope of both the software development and of the control design processes. In fact, since the software has been also developed with the use of control-theory, the domain of the assumptions to test against is also the one of control engineering.

2.2 Control Engineering Primer

In this section, we introduce the definition of a control design problem and the control design process.[1] We illustrate the frequency domain (in contrast to the time-domain) description of signals and systems, and introduce the basic concepts used in the remainder of this work.

Definition of Control Design Problems: As mentioned in Section 1, the objective of the control layer in a CPS is to steer physical quantities to track a desired value. More rigorously, the input of the control layer is a vector of desired (or *reference*) values r. The control objective is to ensure that the actual values in the physical system are as close as possible to the corresponding reference values. Using the control terminology, we say these physical quantities constitute an *output* vector y, and the control objective is $y \approx r$ (i.e., y tracks the reference values in r).

To achieve its objective, the control layer uses sensors to iteratively measure signals from the physical part of the system, and actuators to steer it. Based on measurements and reference values, the control algorithm computes the commands to be

[1] The content of this section is mostly based on the book "Feedback Systems: an Introduction for Scientists and Engineers" [Åstrom and Murray, 2008].

sent to the actuators. The control algorithm is executed repeatedly, at constant time intervals, resulting in a continuous interaction between cyber and physical components. This interaction is called *control loop*; the union of the control layer and physics is called *closed-loop*.

In the drone example, the control objective is to use the propellers to move the drone following a reference trajectory. The software iteratively (i) uses sensors to measure quantities like its own acceleration every millisecond, (ii) executes the control algorithm, and (iii) actuates voltage commands to the motors to spin the propellers. On the physical side, the propellers generate forces that cause the drone movement, and (in turn) affect the future acceleration readings, hence generating the closed-loop interaction.

As r and y are vectors, the engineers usually define multiple control loops, often one for each element of the vectors. In a drone, we can expect to find one loop for each of the three dimensions in the space that the drone can move in: forward or backward, left or right, and up or down. Furthermore, the CPS requirements might call for different control modes, such as fast (but risky) flight mode, and an safe (but slower) one. In traditional control, the design of the different control loops and control modes is addressed separately: engineers develop a dedicated control algorithm for each mode. The implementation can switch between the algorithms of the different modes during execution.

The identification of the control loops and modes is the *control problem definition*, and constitutes a preliminary step in the control design process. For each of the identified loops and modes, the control design problem is the design of the control algorithms needed to control the physics. Each control algorithm, when executed in closed loop with the physical component, is expected to guarantee (to the best degree possible) that the output y tracks the reference r when operating in the mode it is designed for. We now discuss the control design process for one individual loop and mode.

Control Design Process: As highlighted inside the control design block in Figure 1, the design of a control algorithm comprises of three main steps. The first step is to define an equation-based model of the physical component. The role of this model is to provide a representation of how the actuators affect the measurements and output. For example, in the drone, this is a model that represents how the thrust generated by the propellers affects the position and orientation (called attitude) of the drone itself. These models can be obtained using either first-principle approaches, through the laws of physics, or with data-driven approaches through system identification. Either ways, the models are typically in the form of non-linear differential equations, i.e., non-linear equations that contain both signals and their derivatives (rate of change). Non-linear models are generally difficult to analyse, as small changes in the input can cause significantly different behaviours and hence they do not allow for general approaches [Khalil, 2002].

To overcome the complexity of non-linear models, the second step of the control design is to approximate them using linearised models. Retrieving such approximated models is called *linearisation*, and restricts the model scope to the surroundings of an expected operating point.[2] Accordingly, the operating point is chosen as the physical state around which we expect the system to operate most of the time. In the drone, the operating point would be the horizontal state in which the drone is parallel to the ground and not tilted in any direction. Through linearisation we then obtain a model that is still valid for small variations in the attitude angles around the operating point. The model is now a set of ordinary differential equations and therefore it can be handled in a simpler way thanks to a large variety of analytical tools [Åstrom and Murray, 2008].

The third step is finally the design of the control algorithm. For models based on linear differential equations, control theory provides numerous tools to perform exact analyses and design control algorithms with formal performance guarantees. Such tools are based on a frequency-domain description of the physical system. Frequency-domain descriptions are well-suited for treating ordinary differential equations because they provide a compact description for the derivative of a signal with respect to the signal itself. This makes it easier to analyse the physics and draw conclusions on the system's properties. The control algorithms obtained using control theory are also in the form of linear differential equations.

Frequency Domain Descriptions: We now provide an high-level description of the frequency domain for both signals and systems, together with some intuition about why the frequency domain is well suited for treating differential equations.

The frequency domain description of signals is based on the fact that signals can be decomposed and treated as the sum of sinusoidal functions with different frequencies. The description in the frequency domain specifies which sinusoidal components are present in the signal and what their amplitude is. This is in contrast to the time-domain, where signals are represented as a sequence of values over time. The frequency-domain sinusoidal components are commonly called *frequency components*: for example, a fast-changing signal is mostly composed by fast (i.e., high-frequency) sinusoids. On the contrary, a signal that does not change much is mostly composed by slow (i.e., low-frequyency) sinusoids.

The translation of a signal from the time-domain representation to its frequency-domain one uses the *Fourier Transform*—or its time-sampled equivalent *Discrete Fourier Transform* (DFT [Cooley and Tukey, 1965]), which we will use in the remainder of this paper. Figure 3 shows three examples of time-domain signals and their frequency domain representations obtained with the DFT. The first row shows a constant signal, whose frequency representation consists of a single sinusoidal wave at 0 Hz. The second row shows a pure sinusoidal signal, that is mapped by

[2] Linearising a non-linear equation means approximating its non-linear relations (e.g. if a variable is squared, like in the case of aerodynamic drag) with a linear function. The linear function is based on the first derivative of the non-linear relation, and more specifically, on the value of the first derivative in the chosen operating point.

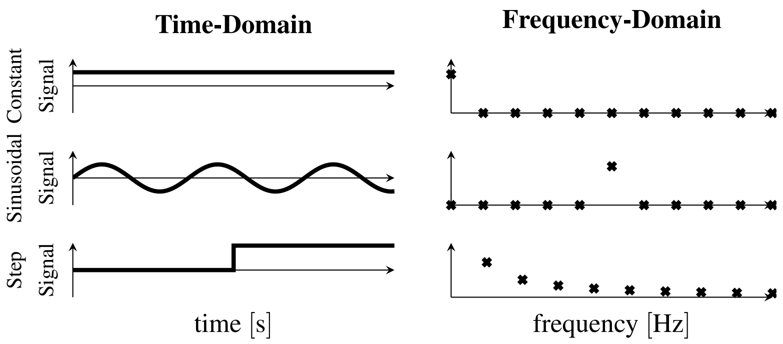


Figure 3. Examples of DFT spectra (on the right) of different signals defined in the time-domain (on the left). A constant signal is described by only a zero frequency component, a pure sinusoidal maps to one single frequency component, and a non-periodic step maps to multiple frequencies.

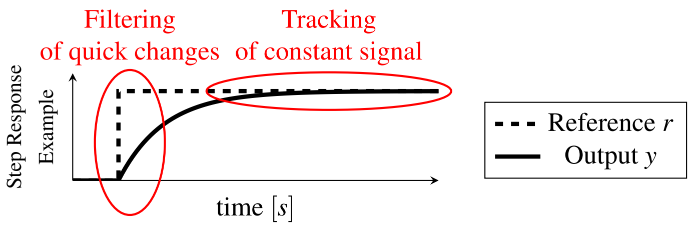


Figure 4. Example of how we can expect the CPS output y (solid line) to track the desired value r (dashed line). In the figure we intuitively highlight how a control system usually behaves like a low-pass filter, by filtering the fast-changing components of the input and tracking the slow-changing ones.

the DFT into a single frequency component. More complex signals, like the step function in the third row, include larger number of frequency components.

The frequency-domain representation provides a compact description of signals according to their the rate of change, or frequency content. The derivative of a signal is another signal that describes its rate of change. This correspondence can be seen as the intuitive reason why the frequency-domain description is convenient for analysing differential equations.

In the frequency domain, systems (i.e., entities that take an input signal and generate an output signal) are described by how much they react to an input according to its frequency content, and more specifically by how much they amplify or reduce every frequency component of the input signal. For example, many physical systems behave like a low-pass filter, transmitting or amplifying slow signals (i.e., low-frequency components), while reducing quickly changing components (i.e., high-frequency components). This behaviour of reducing a frequency component is called *filtering*.

In the case of control systems, ideally we want the physical quantity y to track the reference r at every time instant. In the frequency domain this corresponds to unit amplification between the reference and the output at every frequency. This is apparently infeasible, since a change in r (a variable in the software) requires some time for y (the physical quantity) to follow. In other words, reference changes that are too rapid cannot be tracked by the output of a system. In Figure 4 we show an example of how we could expect the drone to follow a step-like change in the desired position along one direction. The output y, denoted by the solid line, does not follow instantly the reference r, denoted by the dashed line. On the contrary, the quick change of reference value is smoothed in the output signal, which gradually reaches the new reference value. The frequency domain concept of filtering can be used to describe this phenomenon. In fact, we could rephrase this as "the output only tracks the slowly changing (low-frequency) components of the reference, while it filters the fast-changing (high-frequency) ones". We illustrate this intuitive interpretation in Figure 4 with red ellipses. The ellipse on the left highlights the filtering behaviour that occurs when the reference signal has a rapid change, while the one on the right highlights the tracking behaviour when the reference signal does not change.

Given this behaviour consisting of tracking the lower frequencies and filtering the higher frequencies, for a given control loop, we can identify the so-called *closed-loop bandwidth*, denoted by f_b. The closed-loop bandwidth is the threshold frequency below which we expect to have a tracking behaviour (i.e., $r \approx y$) and above which we have a filtering behaviour. This quantity therefore corresponds to the fastest frequency components of the reference that the control system is able to track. Accordingly, it is also considered a quantification of the speed of the control system: the higher the closed-loop bandwidth, the higher the speed of the control system.

While it may seem intuitive that the control engineer wants to design a control algorithm that maximises f_b, to obtain a fast system there are other factors to account for. As an example, a high value of f_b usually comes at the cost of a high control actuation (e.g., a fast-moving drone will generate high forces that can ruin the actuators). Furthermore, noise can be found in the measurements at high frequencies: an accelerometer that measures a drone's acceleration is usually affected by high-frequency electrical noise. If the control system reacts to input signal in the high-frequency range (obtaining high speed), then it will also react to noise. In turn, this will reduce the system performance making its behaviour unpredictable. Such considerations lead to a trade off in the control algorithm design between speed of the system and noise rejection.

2.3 Related Work

The topic of testing control-based CPS software is not new in the software engineering literature. In fact, control systems are seen as a class of CPSs in which most of the added value is placed in the software part [Broy et al., 2007]. Accordingly, recent works highlighted interesting research directions at the intersection of control and software engineering [Balasubramaniam et al., 2020; Bradley and Bagheri, 2020].

Testing of control systems has been approached in the literature in different ways.

Given the widespread industrial use of Simulink[3] model testing has received significant attention in the last decade [Briand et al., 2016], with recent work showing the complementarity of model checking and model testing for the verification of requirements [Nejati et al., 2019]. Generation of test traces is one of the main topics. Very different types of algorithms are used for the generation of input sequences: search algorithms [Matinnejad et al., 2014; Matinnejad et al., 2017; Marculescu et al., 2015], classification trees [Lamberg et al., 2004], system-identification based refinements [Menghi et al., 2019], and search algorithms [Aleti and Grunske, 2015; Hänsel et al., 2011]. We also note a number of application-specific works in the avionics [Peleska, 2002; White, 2001; Samad and Balas, 2003] and automotive [Bringmann and Krämer, 2008; Bringmann and Krämer, 2006] domains.

CPS falsification (as the generation of test cases that falsify a given requirement) is also an active research direction. In a recent work [Yamagata et al., 2021] the authors use deep reinforcement learning to perform robustness guided falsification. Interestingly, they note the importance of making the system internal dynamics available to the reinforcement learning algorithm, which highlights the possible benefits of leveraging control-design information during the testing. Furthermore, a combination of model checking and path planning can be used to invalidate linear-temporal-logic formulas [Plaku et al., 2009]. Similar problems can be addressed with rapid exploration of random trees [Dreossi et al., 2015].

The embedded software literature sees control systems as a prime application of computing in a resource constrained environment [Garousi et al., 2018]. This is due to the presence, in control software, of constraints on non-functional properties. Accordingly, it appears in several surveys and reviews on the testing of embedded software [Zander et al., 2011; Garousi et al., 2018; Banerjee et al., 2016].

The control and robotics communities show growing awareness of the impact of the software implementation of control algorithms on their performance [Zimmer et al., 2015; Silano et al., 2018]. Zimmer et al. [Zimmer et al., 2015] discuss a case study on the consequences of implementation choices for the control performance. In robotics, software-in-the-loop simulations (i.e., simulations that include

[3] A software environment for the development of control systems (`https://www.mathworks.com/products/simulink.html`).

the actual control software implementation) can be effective in exposing bugs in the software design of quadcopter controllers [Silano et al., 2018; Timperley et al., 2018].

None of these works explicitly leverages the control-theoretical design of these systems or integrates control knowledge in the definition of the software testing problem. Partial exceptions are [Aleti and Grunske, 2015; Menghi et al., 2019]: in both cases system identification (a field very close to, if not part of, control theory) is used to reduce the number of tests that have to be executed in order to find faults [Menghi et al., 2019] or to reduce the parameter definition effort required by genetic algorithms [Aleti and Grunske, 2015]. In these approaches, system identification is used as part of the testing process, and not to obtain knowledge on the system under test. In addition, a recent work in the context of runtime verification leverages system dynamics (differential equations, the same type of models used at control design time) to either terminate the monitoring early, or to skip samples of the signal [Abbas and Bonakdarpour, 2022].

In this work we introduce the control engineering perspective starting from the CPS testing problem definition, i.e., with the discussion of the control design assumptions and the consequent limitation of the CPS design scope.

3. Control Engineering Perspective on CPS Stress Testing

In this section, we first motivate and define the problem of stress testing the control layer in a CPS. Second, based on the development workflow of control-based CPS illustrated in Section 2, w identify the classes of assumptions that the engineers make in the different control software development stages. For each of these classes, we discuss which techniques are already available for testing the corresponding assumptions, and which assumptions require an application-specific solution; we exclude the latter from the scope of this work. For the remaining classes we discuss the relative dependencies that can be identified: i.e., we identify which assumption classes should be tested first. We conclude this section defining the problem addressed in this paper.

3.1 Problem Motivation

As discussed in Section 2.2, when applying control theory, engineers make *design assumptions* about both the physical part of the CPS and the control algorithm to be developed. The role of these assumptions is to abstract away the details of the system that are not necessary for development, and to define the fundamental building blocks used by the theory. For instance, when designing the control algorithm of a drone, engineers assume that the generated thrust is proportional to the voltage command and that, for example, it does not saturate when the maximum power of the motors is reached. This allows engineers to use a linear model of how the voltage, when applied to the motors, affects the drone movement and position. This linearity assumption is necessary to apply traditional control theory [Åstrom and Murray, 2008].

As discussed in Section 2.1, software engineers are provided with the control algorithms from control engineers. These algorithms are only one component of the control layer. In fact, when implementing the latter, software engineers address the implementation of other functionalities such as the flight mode changes, the interaction with sensors and actuators (e.g., filtering and sanity checks), the parallel execution of the different control loops and the discretisation of the equations. The implementation and integration of other software functionalities can make the design assumptions made by control engineers become invalid. For example, the linear model used for the control design is an assumption that is falsified when the drone motors saturate. This happens because the motors are requested to produce more thrust than their capacity, as in the cases when the reference value changes too much or too fast. In such scenarios, the drone control algorithm will be operating in conditions different from the ones assumed during the design.

Control algorithms are usually robust (at least to some degree) to the falsification of the different design assumptions; this property is one of the reasons for the successful adoption of control theory [Åstrom and Murray, 2008]. Control theory provides metrics to quantify the algorithm robustness to the deviation from assumptions, e.g., "stability margins". However, those metrics are also based on the control design models and are therefore still subject to the validity of their design assumptions. Hence, *the quantification of the extent to which a CPS can be pushed outside of the validity boundaries of its design assumptions* still requires an empirical evaluation. This evaluation can be obtained through stress testing of the software that implements the control layer, by targeting the control design assumptions.

3.2 Design Assumptions in Control Algorithms

Before defining our stress testing problem, we need to identify the classes of design assumptions that control engineers make at design time. These assumptions are made at the different development stages of a control-based CPS. In Section 2.1 we identified three main development stages:

(i) control problem definition,

(ii) control algorithm design, and

(iii) control algorithm implementation.

We now discuss the design assumptions made in each of the stages.

Assumptions at Control Problem Definition Time: At this stage the engineers identify the different control loops and modes for which they will develop a control algorithm. As a consequence, when designing the individual control loops, they assume that (i) the different loops do not interfere with each other and (ii) the mode changes do not impact the control design performed afterwards. For example, for a drone, the design of the altitude controller may not account for the horizontal controllers (and vice versa). Similarly, the design of the “aggressive flight” controllers is done independently from the “safe flight” controllers. Such assumptions significantly simplify the design of the control algorithms, allowing, among others, to independently design the response to a change in each element of the vector r. However they do not always hold in practice. For example, when the drone tilts to move horizontally, it also loses vertical thrust, affecting the altitude controller. Another case of assumptions not holding is when a mode change command is issued during the flight. Such a change can cause a sudden change in the motors' commands, which possibly affect the CPS performance.

Assumptions at Control Design Time: During the control design, the engineers develop a non-linear model of the physical part of the CPS, also on the base of information received from the physical-component engineers that designed it. Such a model (like any model) is only an approximation of reality and will neglect or approximate certain aspects of the problem. For example, a drone model assumes a given mathematical relation between the rotational speed of the propellers and the generated vertical thrust. However, this type of aerodynamic phenomena are difficult to quantify. Moreover, there could be some inconsistency between the mathematical model and the real physical system. By using such models, the engineers implicitly assume that they are a sufficiently accurate representation of the physical reality.

As mentioned above, the models of the physics also need to be linearised in order to use the tools from control theory. The linearised version of the model is only valid in the surroundings of the operating point chosen for the linearisation. Practically, by using the linearised model, the engineers implicitly assume that, during operations, the CPS stays sufficiently close to the operating point so that the linearised model is an accurate enough representation of the physical part. For example, the propellers cannot generate more thrust than the motors can provide: in fact, the motors saturate (max-out) once they reach their maximum capacity. To linearise this relation, the engineers assume that the motors are not in the saturated state, and that they always provide a thrust proportional to the voltage command. When, during the actual flight, the motors saturate, this proportional relation loses validity (and therefore also the model and design).

Assumptions at Control Algorithm Implementation Time: Control algorithms are generally specified as linear differential equations. Such equations are defined with the use of continuous mathematics. However, they are implemented on computers which are discrete machines. Hence they have finite precision in the representation of the parameters and execute the algorithms in discrete steps over time. Accordingly, the engineers, when designing the control algorithm with continuous mathematics, are implicitly assuming that the discretisation happening during the implementation does not significantly alter the algorithm (with respect to the ideal mathematical object). More specifically, they assume both that the finite precision does not significantly alter the computed values, and also that the discrete execution does not alter the frequency properties (meaning the properties of the algorithm execution over time).

Design Assumptions Summary: To summarise, we identify the following classes of design assumptions that are made by the engineers during the development of control algorithms:

- they neglect the interaction between different control loops;
- they neglect the impact of mode changes on the control algorithms performance;
- they assume that the non-linear model of the physics is a sufficiently accurate representation of the real system;
- they assume that the system stays sufficiently close to the operating point chosen for the linearisation;
- they assume that the finite precision of the representation of the equation variables and parameters is sufficiently accurate; and
- they assume that the execution in discrete time steps does not significantly affect the expected execution time properties of the algorithm.

When performing stress testing for a control-based CPS, engineers should aim at falsifying each of these assumptions.[4] We now discuss for which classes of assumptions there exist already software testing techniques and which ones require an application-specific solution.

Testing numerical properties of numerical algorithms is not a novel problem; there is a significant literature corpus [Yi et al., 2017; He et al., 2020], also targeting control algorithms [Sanchez-Stern et al., 2018; Magnani et al., 2021]. Similar considerations can be made about test the execution timing properties. A number of works can be found in the literature for testing embedded software (characterised by the relevance of non-functional properties like execution timing) [Garousi et al., 2018; Afzal et al., 2009]. Furthermore, we note recent works dedicated to the verification and testing of the robustness of control algorithms to execution timing faults [Ghosh et al., 2022; Vreman et al., 2021]. Given the above previous works, we leave the testing of numerical and timing properties out of the scope of this paper.

[4] We note that there are branches of control engineering that try to mitigate each of those simplifying assumptions, e.g., multivariable control and robust control. However, like the stability margins mentioned above, such approaches are still subject to design assumptions and also require verification. Furthermore, those are rather advanced theories and, as of now, find limited application in practice [Desborough and Miller, 2002].

Testing the validity of the physical model is a highly application-specific problem. To test the aspects of the physics model that are unknown one must know the aspects that were uncertain when developing it. For example, for a drone, two assumptions of the model can be on the aerodynamic properties of the propellers (needed to evaluate the vertical thrust that can be generated) and on the rigidity the drone body (to simplify the equations describing the motion of the drone in space). Among those, the former is likely to be associated to a higher degree of uncertainty because the aerodynamic phenomena are generally hard to characterise. In contrast, the assumption on the rigidity is more likely to be valid: intuitively, we do not expect the drone motor supports to bend. Such considerations are clearly application-specific and require an understanding of the specific model that is being used. Accordingly, the generation of test cases that falsifies this type of assumptions cannot be treated in a general fashion. Given its application-specific nature we leave the testing of this type of assumptions out of the scope of this work.

We are therefore left with the assumptions regarding non-interactions between control modes and control loops, and about the sufficiently large range of validity of the linear models. Among those we note that the first two are dependent on the latter. In fact, if an individual control loop does not have a sufficiently large range of validity when operating independently (i.e., without mode switches and in absence of reference changes for the other loops), then, the switching across different modes and the interaction between loops are unlikely to improve its range. For example, if we have an altitude control loop for a drone that is not very robust when operating alone, then it is unlikely to perform better when the control loops of the horizontal directions are also active and can disturb it. Given this dependency, we argue that *testing the validity of linear models should occur before testing the interactions between control modes and control loops*. In light of this discussion, this work focuses on the testing of the linearised model control design assumptions, for which we give our problem statement below.

3.3 Problem Statement

In this work, we address the problem of stress testing the linearised model design assumptions in an individual control loop of a CPS control layer. The *objective* is to generate and identify tests that create a gap between the behaviour of the system and the linearised model used during the control algorithm design. This gap should appear in different degrees and make the control algorithm increasingly unable to provide the control-theoretical guarantees. Accordingly, our *test inputs* exercise the control layer and consist of sequences of reference values over time. Our *test outputs* are the traces of the physical quantity that has to track the reference value.

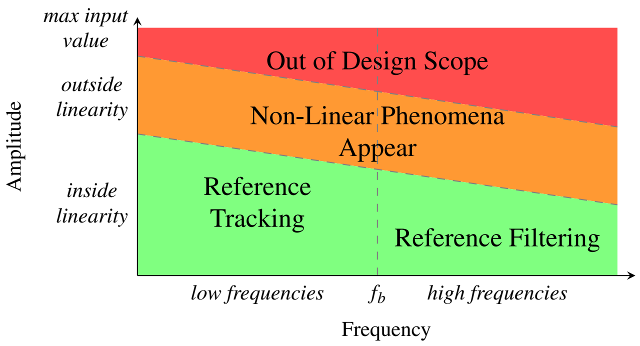


Figure 5. Qualitative frequency-amplitude characterisation of the input space of a signal control loop. The colours highlight the validity of the linearised model with respect to the input frequency content and amplitude The green area corresponds to the input signals for which the system remains within the assumptions of control theory. The orange area corresponds to the input signals that trigger non-linear phenomena but not enough to cause significant performance degradation in the system. The red area are the input signals for which control theory assumptions are not fulfilled and the behaviour of the system becomes unpredictable.

4. Control Loop Input Space Characterisation

In this section, we use domain knowledge from control theory to provide a *qualitative characterisation* of a single control loop input space. The proposed characterisation maps frequency and amplitude features of the input to the expected behaviour, i.e., the expected relation between the (scalar) output y of a control loop and the (scalar) input reference r. In the first part, we present the qualitative characterisation based on the validity boundaries of the linearised model and insights from control theory.[5] We use a minimal example (a simplified model of the altitude control of a drone) to exemplify the different system behaviours highlighted by the characterisation. In the second part of this section, we list the qualitative aspects of the characterisation and propose approaches to quantify them. Such quantifications enable the practical use of the characterisation, and constitute the basis for the test case generation approach proposed in the following section. We conclude the section discussing our problem statement in the context of the proposed characterisation.

[5] We note that a similar qualitative characterisation of the input space of a control loop is found only in one book on control engineering from 1959 [Gille-Maisani and Decaulne, 1959]. However, the treatment of the topic is brief and high-level and has not been investigated further in later literature.

4.1 Qualitative Input Space Characterisation

In order to leverage domain knowledge from control-theory (Section 2.2), we base our characterisation on a frequency-domain description of the input sequences. Practically, this means that we describe the input space with two dimensions: one captures the input frequency content and the second captures its amplitude. Being a two-dimensional space, the input space can be represented as a *frequency-amplitude plane*. We provide a graphical representation of this input space plane in Figure 5: one input sequence corresponds to one or more points according to its frequency content and its amplitude. We now use control-theory knowledge to identify different areas in the input plane according to the expected behaviour of the control loop, depicted by colours and boundaries in Figure 5). We now identify these areas in terms of:

- where control theoretical guarantees apply (i.e. validity of the linear model), and
- tracking and filtering behaviour within the applicability boundaries of control theory.

In order to exemplify the different behaviours that we highlight in the characterisation we use a minimal example of the altitude control of a drone. To enable the easy detection of the limitations of the linear model, we use a simulator based on a linear model and introduce one single source of non-linearity: the saturation of the thrust. Practically, this saturation limits the force that can be applied by the motors to move up and down the drone. In Figure 6 we report four tests showing the response to square waves with different amplitudes and periods. For each test, the upper plot shows the desired position (the black line) and the actual position (the blue line). The lower plot shows instead the command sent to the motor that can be used to accelerate or decelerate the drone (the red line). The saturation of the motor (and hence the validity of the linear model) can be detected in these plots when the force becomes fixed at $\pm 2\,\mathrm{N}$.

Within Applicability of Control Theory: Inputs of small amplitudes will not push the CPS far away from its operational point. Accordingly, for lower amplitudes we are within the validity bounds of the linear model: this is represented by the green area in Figure 5. Within this area, we expect the system to be able to track the slower inputs: those sequences correspond to low-frequency inputs inside the "***Reference Tracking***" area. Faster signals map instead to higher frequencies and are not expected to be tracked: those belong to the "***Reference Filtering***" area. In the figure we highlight the closed-loop bandwidth f_b that separates the tracking and filtering areas. To exemplify the tracking and filtering behaviours, in the upper plots of Figure 6, we feed the controller of the drone with a slower (in the left-hand side plot) and a faster square wave (in the right-hand side plot), both of amplitude $0.6\,\mathrm{m}$. In the former we can see that the reference is successfully tracked within seconds

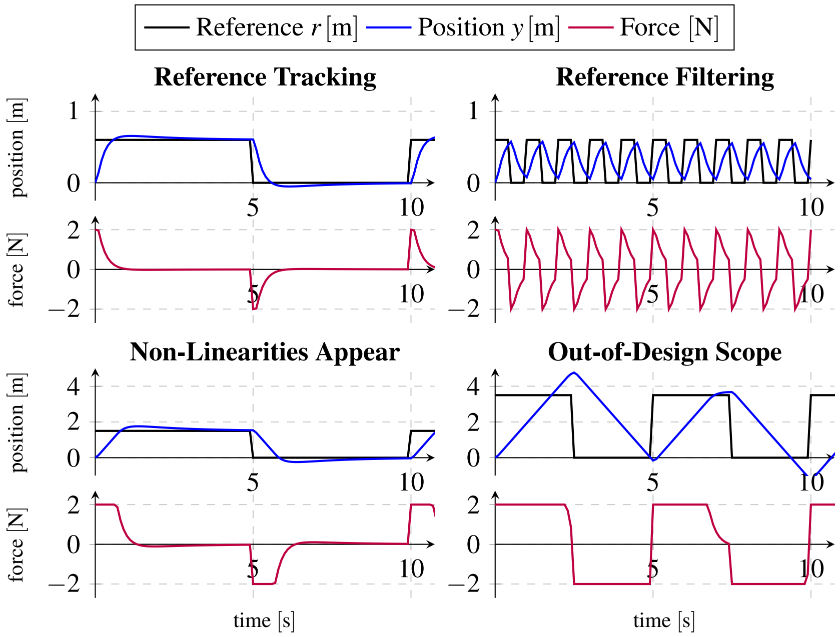


Figure 6. This figure exemplifies the four different behaviours highlighted in the qualitative characterization of the input space of a control loop. The plots are based on the simulation of a minimal model of the altitude control of a drone. The only source of non-linearity in this example is the saturation of the motor that limits (between ± 2 N) the thrust that can be generated by the motors. The blue lines show the altitude of the drone (i.e. the output y), the black line shows the desired altitude (i.e. the input r), and the red line in the lower plots shows the thrust generated by the motors. Appearance of non-linear phenomena can be detected when the red line saturates at ± 2 N.

after a step. In the latter the reference changes are too fast and the drone cannot follow it successfully: we say therefore that it is filtered.

Validity Bounds of Linearised Model: When we consider input signals that are larger in amplitude, the CPS moves further away from the operational point used for the linearisation and non-linear phenomena start to appear. In Figure 5, such signals correspond to the orange area "***Non-Linear Phenomena Appear***". In the altitude controller example this corresponds to hitting the motor saturation. Accordingly, in the lower-left plot of Figure 6, we feed the drone with a larger square wave of amplitude 1.5. As we can see from the plot of the control action, the motor now saturates for some time after the occurrence of the step in the reference. This, however, does not significantly affect the way that the actual altitude of the drone follows the desired reference, i.e., the reference is still successfully tracked. In other words, the control algorithm is showing some robustness to the motors being saturated over a limited amount of time. When the system moves even further away from the design scope, the linearised models are falsified and there is no way to predict the system behaviour. This is the "***Out-of-Design Scope***" red area. In fact, in the lower right plot of Figure 6, we can see that the drone is not only unable to track the square wave of amplitude 3.5 m, but it also exhibits a new behaviour, a triangular wave.

Finally, we note that either large (high amplitude), fast changing (high frequency) inputs, or a combination of the two can lead the system out of its design scope. For example, in the drone altitude control, either a fast-changing input or a large input can require high thrust and hence can cause motors saturation. Accordingly, we draw the thresholds for which non-linear behaviour appears and show the bound of the design scope to be decreasing with respect to increasing frequencies.

4.2 Qualitative Aspects of the Characterisation and their Quantification

We aim at using the qualitative characterisation proposed in the previous section to generate test cases that invalidate the linearised model used for the algorithm design and hence push the control layer to its performance limits. In other words, we want to be able to sample (test) points in the frequency-amplitude plane and identify the behaviour that the test results expose in various areas of the plane. In order to stress test the CPS, we want to sample around the border of the "out-of-scope" area to understand when the control algorithm is no longer able to provide the performance guarantees. Furthermore, we also want to identify the border between "tracking" and "filtering" behaviours in order to characterise the fastest signals that the control algorithm can track.

We note that the latter distinction between the "tracking" and "filtering" areas (i.e., the closed-loop bandwidth) is not strictly related to the falsification of the design assumptions (both areas are in fact coloured green). However, it represents how fast a reference the control loop can track and hence represents a performance limit of the system. If we want to push the system to its performance limits, then we have to ensure that the test cases cover both behaviours.

Accordingly, in order to make our qualitative characterisation usable for the generation of stress test cases, we have to make quantitative the following qualitative aspects:

- One input sequence generally contains more than one frequency, and hence can map to more than one point in the frequency-amplitude plot. Accordingly, we need to define a *mapping between a test input sequence to a corresponding set of frequency-amplitude coordinates* in the input plane. This enables the identification of which points in the frequency-amplitude plane are sampled by a test.

- The detection of when a test trace shows a behaviour that is out of the validity ranges of the linearised model—i.e. when a test belongs to the red area—does not have a formal definition in present literature. Accordingly, we need to define a "*degree of non-linearity*" observed in a given test result. This enables the detection of test cases that belong to the red area and that are outside of the scope of the design assumptions.

- Since a test can map to multiple frequency-amplitude points, it can expose multiple behaviours simultaneously. Accordingly, we need to define a *mapping between the different behaviours observed in a given test and its frequency-amplitude points* (i.e. the different coordinates mentioned above). This enables the distinction of the different behaviours (tracking, filtering, out of scope) that might appear in the same test.
- It is practically impossible to know a priori the actual shape of the threshold for which non-linear phenomena start to appear, nor of the threshold for which they invalidate the linear models enough to impair the system performance. Hence, the bounds between the areas of different colours can have an arbitrary shape. However, we can *define a set of properties that are generally expected to hold between the behaviours exposed by test cases in different points of the input plane*. This enables the definition of test case generation strategies in the frequency-amplitude plane that explore the different behaviours of the control loop. Furthermore, said properties can be used as sanity checks for the testing process.

We now address the definition and quantification of each of those qualitative aspects. Since we leverage the DFT of the input and output of the tests, we report in Figure 7 the frequency-domain representation for the tests of Figure 6. The figure uses the same colour convention as its time-domain equivalent: blue crosses represent the frequency components of the trace of the actual position of the drone, and black crosses represent the frequency components of the input sequence. We use the plots of the DFT to exemplify the different definitions and explain the underlying intuitions. We remark that, analogously to common practice in the frequency-domain, we use a logarithmic scale on both the axes of all the plots in Figure 7. This enhances the readability of the plots.

Mapping of Tests to Frequency-Amplitude Points: Given an input reference sequence $r(t)$, we want to define the frequency-amplitude coordinates that we are sampling with the associated test. We define this mapping according to the frequency spectrum (the DFT) of the input reference. In practice, inputs are signals sampled over time, and therefore also the spectrum computed with the DFT is discrete [Cooley and Tukey, 1965]. More specifically, the time-domain samples are mapped to an equally numerous set of equally spaced frequency components, like in the DFT examples in Figure 3. The number of the frequency components is therefore large: for example, a 5 seconds trace sampled every millisecond is mapped to 5000 frequency components. However, most of those components are usually zero or close

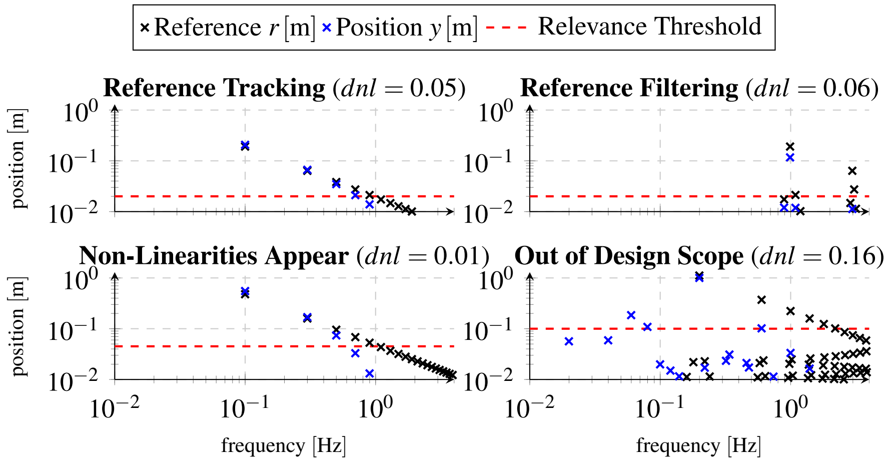


Figure 7. This figure reports the input (the black crosses) and output (blue crosses) DFT of the traces from the tests on the minimal example of the drone altitude control shown in Figure 6. These plots exemplify how we use the frequency content of the input and output to detect the different behaviours of control systems. The filtering behaviour is detected when input components are not found in the output (upper-right plot). Non-linear behaviour is detected when new frequency components are found in the output (bottom-right plot). Furthermore, the red dashed line highlights the threshold that we use to identify the relevant input components use for the mapping of the test case to the frequency-amplitude plane.

to it, meaning that only few of those samples actually carry information about the signal.[6] Accordingly, among all of the frequency components computed with the DFT, we consider as relevant only the ones with larger amplitudes. Formally, given an arbitrary input $r(t)$, we map it to a set of frequency amplitude coordinates (f,A):

$$fAmap[r(t)] = \{(f,A) : A = |DFT[r(t)](f)| \wedge A > \gamma \cdot \max_f \{|DFT[r(t)](f)|\}\}, \quad (1)$$

where $DFT[\cdot]$ denotes the DFT, $|\cdot|$ denotes the modulus,[7] and γ is a parameter in the range $[0,1]$ that we use to select the relevant components (i.e. the larger ones) in a relative way to the largest one: $\max_f\{|DFT[r(t)](f)|\}$. We exemplify this approach to selecting the relevant points of the input DFT with the red dashed line (for $\gamma = 0.1$) in the plots of Figure 7. In fact, the quantity $A > \gamma \cdot \max_f\{|DFT[r(t)](f)|\}$

[6] The reason for excess of samples is that signals are usually oversampled (sampled more frequently than strictly necessary) for redundancy and robustness. Such oversampling introduces extra frequency components in the higher part of the spectrum that do not carry much information about the signal and are therefore zero or close to it.

[7] The modulus is needed as the DFT computes the frequency components as complex numbers.

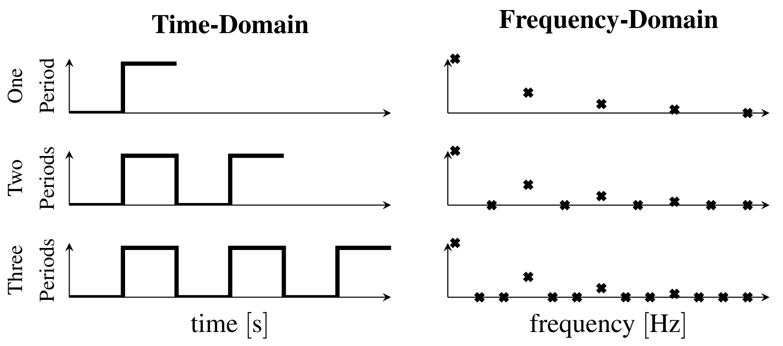


Figure 8. This figure shows a graphical representation of how the repetition of the same input sequence increases the resolution in the frequency-domain. The repetition of the same sequence does not introduce new information to the input signal, whose non-zero frequency components remain unchanged. However, when the input is repeated, the output signal is now sampled also in frequencies outside of the input main components (the ones that are null in this plot) hence enabling the detection of frequency spectrum broadening because of non-linear behaviour.

defines a threshold above which we include the components and below which we exclude them. Accordingly, the f/A points of the input space sampled by the tests are the are the black crosses found above red thresholds—i.e. the larger components of the DFT of the input sequence. We can then observe in Figure 7 that: the faster square waves of the two right-hand side plots map to points further to the right on the frequency axis with respect to the other tests associated to slower square waves. Furthermore, the larger amplitudes of the square waves from the bottom plots map to points higher in the amplitude axis with respect to the other tests associated to smaller square waves. This exemplifies how the size and the speed of the inputs are captured in the frequency-domain.

Degree of Non-Linearity Definition: As exemplified in the bottom right plot of Figure 6, non-linearities affect the CPS by introducing in the output new components that were not present in the input. This kind of behaviour can be harmful as it implies that the control algorithm is introducing some new behaviour in the system that was not part of the reference. For example, in the bottom right plot of Figure 6 the altitude of the drone reaches 4 m, when the reference is at 3 m. This intuition of non-linearities introducing new components finds confirmation also in the telecommunication field. In fact, non-linear components in telecommunication electronics are known to cause the the broadening of the spectrum of a signal [Smirnov et al., 2006]. Broadening of the spectrum means that in the output we find frequency components that were not present in the input signal.[8] The bottom right plot of Figure 7

[8] From a theoretical perspective, this can be explained in terms of the Taylor expansion of a non-linear

shows the DFT of the input and output of the test that shows non-linear behaviour in our drone altitude control example. In this example, we can note the presence in the output (the blue crosses) of components in the frequency range $[0.02, 0.4]$Hz that were not present inn the input (the black crosses). This is the broadening of the spectrum mentioned above.

According to this intuition, we define the degree of non-linearity on the base of how much output signal we find outside of the input spectrum. This requires that we *sample the spectrum of the output also outside of the frequencies of the input components*. In order to increase the number of samples in the frequency spectrum without altering the frequency content of the input we can repeat the input sequence (i.e., make it periodic). This apparently does not alter the information contained in the input (since it is just repeated), and the non-zero frequency components do not change. However, for each repetition we double the samples in the time-domain, hence also double the samples in frequency-domain. The new samples obtained in this way are found on the frequency axis *between* the samples previously available (rather than only in the higher part of the spectrum, as for the samples introduced by the oversampling). We exemplify graphically the sampling of new frequencies in Figure 8. The figure shows how repeating the steps in the time-domain (plots on the left-hand side) increases the resolution in the frequency-domain (plots on the right-hand side) by adding new samples outside of the spectrum of the non-repeated input (the first row).

When identifying non-linear behaviour, we are interested in detecting *new frequency components that are too large*. We are not interested, for example, in how many those are, since even a single one can be harmful. Hence, we define the degree of non-linearity according to the *maximum amplitude* of the output spectrum outside of the main components of the input spectrum. We look therefore at all the frequencies present in the DFT of the output y minus the relevant ones found in the input (i.e. $fAmap[r(t))]$). Formally, given a reference sequence $r(t)$, we can define the set of frequencies to check as $f_{new} = \{f : \exists DFT[r(t)](f) \wedge \neg(f \in fAmap[r(t))]\}$ that takes all the frequencies of the input DFT and then excludes the ones of the main components. For example, in the tests of Figure 6 it means that we are looking at the frequencies that are *not* associated to any input frequency component (black cross) above the red dashed line—i.e., the ones associated to the components below the threshold. Furthermore, in order to obtain comparable results across tests with different amplitudes we normalize our metric with respect to the amplitude of the input $\max_f\{|DFT[r(t)](f)|\}$. We then obtain the following definition for the **degree of**

input-output relation: if the relation is non-linear, then higher-order terms will appear in the expansion. Higher order terms depend on powers higher than 1 of the input, which is then multiplied by itself. Then, even considering the simplest case of a sinusoidal input we can see with basic trigonometric rules that new frequencies appear—e.g. the sinus squared makes appear a new component at double the frequency: $2\sin^2(x) = 1 - \cos(2x)$.

non-linearity of a given test i

$$dnl(i) = \frac{\max_{f \in f_{new}} \{|DFT[y_i(t)](f)|\}}{\max_f \{|DFT[r_i(t)](f)|\}}, \tag{2}$$

where apparently $r_i(t)$ and $y_i(t)$ are respectively the input and output associated to the test and the other elements follow the same conventions as in previous equations. When we apply the *dnl* formula to the examples in Figure 7, we obtain the values reported in the titles of the different plots.[9] By comparing the numbers, we see that the new frequencies that appear in the bottom right plot cause the *dnl* to be of one order of magnitude higher than the *dnl* of the other tests. This example also shows that already a seemingly small value like 16% of the maximum input amplitude can significantly impact the performance.

To conclude, we note that the idea of repeating the input sequence exposes the trade-off between test duration and the frequency resolution. Naturally, higher frequency resolution increases the chances of detecting new frequency components, hence non-linear behaviours. However, more repetitions require longer tests. How many frequency samples are needed to detect new frequencies in the output is dependant on the specific application. In the experimental part of this work, we run tests preliminary tests to explore this trade-off for our case of studies and select an adequate number of input repetitions.

Mapping of Behaviour to Frequency-Amplitude Points: When we analyse the degree of non-linearity, we obtain a metric that characterises the whole input sequence. Said in other words, it characterises equally each of the input frequency-amplitude components. In fact, for example, there is no way to identify which input component in the bottom right plot of Figure 7 is causing the non-linear behaviour. Hence, we associate the *dnl* metric equally to all of the frequency components of the input.

Differently, when the SUT behaves linearly, some frequency components of the same input are tracked and pass through to the output, while other components are filtered. For example, in the upper right plot of Figure 7 we can observe that the input component at frequency 1 Hz is found, albeit reduced, also in the output, while the component at frequency 3 Hz has much lower amplitude in the output. Therefore, when we quantify the filtering behaviour, first we have to consider only tests that expose linear behaviour, and second we have to analyse the different frequency components individually.

For each of the frequency-amplitude points of the input, we define a *degree of filtering* depending on how much of the input is found in the output. If a frequency component is perfectly tracked, its intensity in the input and output are equal, hence their ratio measures to 1. Instead, if the ratio of the input over the output is below the unit, it corresponds to filtering since part of the signal is lost. Analogously, values

[9] In this cases, to compute the *dnl* we used ten repetitions of the step sequence.

above 1 detect amplification: while a small amplification can be expected in real-world systems, large amplification can be dangerous (for the very same reason as the risks of the broadening of the frequency spectrum).

Hence, given a test i, we define the degree of filtering *for a given input frequency component* f as the difference between 1 and the mentioned output-input ratio:

$$dof(i,f) = 1 - \frac{|DFT[y_i(t)](f)\,|}{|DFT[r_i(t)](f)\,|}, \tag{3}$$

with the same conventions as for the equations above, and the remark that this definition is valid only for tests with linear behaviour. Given the absolute values at the numerator and denominator (which therefore can only be positive numbers), this metric takes values in the range $[-\infty, 1]$. A value of 1 describes complete filtering, while 0 describes perfect tracking. Negative values correspond to input amplification.

We can then use this degree of filtering to identify the closed-loop bandwidth. According to its control-theoretical definition, the closed-loop bandwidth f_b corresponds to a ratio of 0.5 between output and input [Åstrom and Murray, 2008]. Therefore, the bandwidth can be identified as the threshold below which the frequency components show a dof lower than 0.5 (reference tracking area), and above which the dof is higher (reference filtering area).

Expected Properties of the Characterisation: While the plot in Figure 5 is qualitative, and we cannot predict the shape of the different behaviour areas, we can define some properties that are expected to hold across different tests. Said properties describe the expected relative positioning in the frequency-amplitude plane of the tests showing different behaviours. Therefore, they can be leveraged both to develop test case generation strategies and or as sanity check for the testing process.

From the relative positioning of the different behaviour areas, we can identify the following expected properties:

PR1 non-linear degree should increase for increasing amplitudes and frequencies. In fact, the further we move away from the origin of the frequency-amplitude plane the closer we should be to the input area outside of the design scope.

PR2 for linear tests, the filtering degree should increase as the frequency increases. In fact, faster signals should be always harder to track than slower ones.

PR3 the closed-loop frequency bandwidth should be independent of the specific test. In fact, when the system is behaving linearly, the threshold between the tracking and filtering areas should not depend on the specific input and be instead a property of the system.

We now exemplify an evaluation of these properties on the tests of Figures 6 and 7. Property **PR1** is fulfilled “on the high-level” since, the test with largest amplitude (the bottom-right one) is the one with the largest *dnl* and the one with the second highest *dnl* is the one with the fastest input (the upper-right one). However, when we compare the two plots on the left-hand side we would expect that the lower one has a higher *dnl* than the upper one, since the latter receives a smaller input with the same frequencies. This showcases that, when the linear model loses validity (in this case because of the appearance of the saturation) the behaviour of the system becomes unpredictable and not necessarily worse. Such unpredictability further underlines the importance of testing in the areas of the input space that are at the bounds of validity of control theory. Property **PR2** is fulfilled: looking at the tests that show linear behaviour we can observe that the blue crosses move further down from the black ones as we move to the right (hence to higher frequencies). This means that the reference signal is found in the output less and less. Concerning Property **PR3**, we can observe, in the upper tests, that the frequency above which the input is filtered in the output, is similar for both tests (around $0.9\,\text{Hz}$). For the bottom right test it seems to be a bit lower instead (around $0.6\,\text{Hz}$). This is possibly due to the appearance of the saturation that limits how large and fast references the drone can track (hence practically decreasing the closed-loop bandwidth for larger amplitudes). Given that in the real world we would not necessarily know that the bottom-left test is triggering some non-linear phenomenon, this discrepancy from the two upper tests can be used to highlight that this test might require a more detailed analysis (even though the control performance is possibly still acceptable).

4.3 Benefits and Limitations of the Frequency-Amplitude Characterisation

Our testing objective is to generate and identify stress test cases that push the system around the limits of validity of the linearised model. In the context of our proposed characterisation, this can be quantified as generating test cases that expose different *dnl* values. In fact, different *dnl* values correspond to different levels of non-linear behaviour. More specifically, in order to explore the *validity limits*, we are interested in test cases where the *dnl* is non-zero (hence not being fully within the design scope) but also not too large (hence not being far outside of the design scope).

However, the identification of a boundary in the input space where the *dnl* transitions from the zero value to non-zero values is not possible for arbitrary input sequences. In fact, as discussed above, the *dnl* metric characterises equally all of the frequency-amplitude components of a given test. Consequently, a given point in the frequency-amplitude plane can express different values of *dnl* depending on the other components of the input. For example, in both the two right-hand side tests presented in Figure 7, we can identify a main component around $1\,\text{Hz}$ and amplitude $0.2\,\text{m}$. The two components are very close in the plane but are associated to two test with different *dnl* values (and accordingly different behaviours). The test in the upper plot shows linear behaviour and is within the design scope, while the lower one is outside of the scope. This discrepancy appears because the two components belong to different tests and are therefore coupled with other different components. Practically, this further remarks the qualitative nature of Figure 5 since the highlighted areas cannot be distinguished in the general case.

To overcome this limitation, in our testing approach we propose a novel test-case parametrisation for the control layer of CPSs. With this parametrisation we separate, in a test case, the definition of

- the frequency content,
- the amplitude content and
- the combination (i.e. the relative positioning) of the different frequency-amplitude components.

As shown in the next section, thanks to this separation, we can circumvent the problems that arise from the fact that different inputs can have different combinations of frequency components.

5. Testing Approach

In the first part of this section we propose a novel test case parametrization for the control layer of CPSs. The objective of the proposed parametrisation is to enable and ease the use of the definitions proposed in the previous section. More specifically, it allows for

- the use of the dnl (which requires periodic input sequences),
- the separation of the different input features (frequency content, amplitude content and the combination of different frequency-amplitude components), and
- the definition of the expected properties as metamorphic relations [Ayerdi et al., 2021; Chen et al., 2018].

In the second part of this section, we use our test case parametrisation to develop our testing approach. In our testing approach, we use the dnl and the expected property **PR1** to obtain a preliminary optimistic bound on the input amplitudes that the control layer can track. Using the preliminary bound, we use random sampling to explore the frequency-amplitude plane. Afterwards we execute the test cases, and use the dnl, the dof and the metamorphic relations to sanity check the test outcomes and identify the stress test cases.

5.1 Test Case Parametrisation

One test case i corresponds to a sequence $r_i(t)$ of reference values over the time t. In order to achieve the desired properties listed above, we define $r_i(t)$ as a function defined by three elements: an amplitude gain, a time scaling coefficient and a periodic shape function. More specifically, we use the expression

$$r_i(t) = \alpha_i \, \phi_i(\gamma_i t)\,,$$

where:

- $\phi(\cdot)$ Is a periodic function defining the input shape. For example, it can define a square wave or a triangular wave (arbitrary sequences can be chosen as long as they are periodic). Without loss of generality, we assume that shape functions are normalized to have a unit period (i.e. $\forall t : \phi(t) = \phi(t+1)$) and also have a unit amplitude range (i.e. $max_t \phi(t) - min_t \phi(t) = 1$).
- α Is a gain used to scale the input amplitude.
- γ Is a time scaling coefficient that changes how quickly we go through the input shape.

One test case i is therefore fully defined by a triplet $(\phi_i, \alpha_i, \gamma_i)$. We note that, as long as ϕ is a periodic function, the sequence $r_i(t)$ is also periodic. Hence it is a repeated sequence and enables the use of the proposed *dnl*.

Sampling of Frequency-Amplitude Coordinates: We now use the reference square waves from the examples in Figures 6 and 7 to exemplify how the proposed parametrisation works and how it enables intuitive sampling of the frequency-amplitude plane. More specifically, how it separates the choice of the relative positioning of the different main frequency-amplitude components and of their frequency and amplitude values. The four tests from the figures can be defined using our parametrisation with ϕ being a square wave with unit period, and switching between the values 0 and 1. Concerning the α value, it corresponds to the amplitude of the square wave. Hence, the tests in the upper plots have both amplitude gain 0.6, while the lower plots have amplitude gain 1.5 and 3.5. Concerning the values of γ, the left-hand side plots have period 10, hence a time scaling of 0.1 to make the signal slower. The right-hand side plots have instead periods 1 and 5, hence respectively a time scaling of 1 and 0.2.

The shape function ϕ defines the relative positioning of the different (f,A) test coordinates. Said in other words, it defines their pattern (e.g., combination of larger and smaller components) in the frequency-amplitude plane. Apparently, different shape functions will map to different patterns of the frequency-amplitude components. However, thanks to the linearity of the DFT, this pattern is independent of the α and γ coefficients. For example, if we look at the main frequency-amplitude components of the inputs in Figure 7 (the black crosses above the red dashed line), we can see that the different square waves all map to components along a decreasing straight line.[10] This showcases that relative positioning of the main components is

[10] Square waves map to a decreasing straight line in logarithmic scale. In linear scale they map to an hyperbole, like exemplified in Figure 8.

defined only by the shape function and is independent of the scaling coefficients α and γ.

The α coefficient enables the movement of the (f,A) test coordinates along the amplitude axis: i.e., the choice of the vertical coordinate A. Intuitively, increasing its value makes the frequency-amplitude components map to components further up in the plane. Vice-versa decreasing its value makes them map to components further down. Thanks to the linearity of the DFT, doubling (or equivalently for any other scaling) the value of α doubles the main components' amplitude coordinate A [Cooley and Tukey, 1965]. For example, comparing the two left-hand side plots of Figures 6 and 7 we can see that the two square waves with the same period map to components with the same frequency but scaled amplitude.

The γ coefficient enables the movement of the (f,A) test coordinates along the frequency axis: i.e., the choice of the horizontal coordinate f. Analogously to the amplitude, increasing its value increases the speed with which we go through the shape function, hence increases its frequency content. Vice-versa decreasing its value makes them map to components at lower frequencies. Noticing again the linearity of the DFT, a scaling of this coefficient corresponds to an equivalent scaling of the main components' f coordinate. For example, comparing the two upper plots of Figures 6 and 7 we can see that the two square waves that have the same amplitude but different periods map to points with the same amplitude but respectively lower and higher frequency.

Definition of Metamorphic Relations: The properties introduced at the end of Section 4.2 describe relations between different tests. Properties that concern the inputs and outputs of multiple test cases are known as metamorphic relations (MR) [Chen et al., 2018]. We define one MR for each property. We define the first two MRs as implications over tests with the same shape. Accordingly, we define each on the base of a *condition* and an expected *implication*. The conditions concern relations between the amplitude and frequency content of different test inputs as well as whether the output shows non-linear behaviour or not. The implications concern relations between the *dnl* and dof observed in the tests outputs. Differently, we define the third MR as a property across tests with different shapes.

Using our test case parametrisation, for tests based on the same shape, we can directly identify relations in the input amplitude and frequency content on the base of the parameters α and γ. More specifically, given two tests i and j with the same shape (i.e. $\phi_i = \phi_j$), the relations between α_i and α_j, and γ_i and γ_j identify their relation in terms of respectively amplitude and frequency content. For example, $\alpha_i > \alpha_j$ automatically implies that the test i maps to points all higher in the frequency-amplitude plane with respect to the points of j. Analogously, $\gamma_i > \gamma_j$ automatically implies that the test i maps to points all further to the right in the frequency-amplitude plane with respect to the points of j.

Property **PR1** states that an increase in amplitude or frequency content of the input should cause an increase in the *dnl*. Thanks to our characterisation, for two tests i and j based on the same shape ($\phi_i = \phi_j$) an increase in amplitude and frequency are identified with $\alpha_i > \alpha_j$ and $\gamma_i > \gamma_j$. Their conjunction can be used to define the MR condition. The MR implication instead concerns an increment in the degree of non-linearity. This can be expressed as $dnl(i) > dnl(j)$. Hence, we can write the following MR:

$$\textbf{MR1:}\ (\phi_i = \phi_j) \wedge (\alpha_i > \alpha_j) \wedge (\gamma_i > \gamma_j) \implies dnl(i) > dnl(j)\,. \tag{4}$$

Property **PR2** states that, as long as the tests shows linear behaviour, an increase in the frequency content will correspond to increase in the filtering behaviour. To identify linear tests we use a threshold dnl_{th} on our degree of non-linearity. Tests below said threshold ($dnl(i) < dnl_{th}$) are therefore considered to show linear behaviour and we limit the definition of this MR to those. Like for the previous MR, we identify an increase in the frequency content for two tests with the same shape as a greater time scaling parameter $\gamma_i > \gamma_j$. The conjunction of the dnl threshold and the increase in frequency content constitute the MR condition. For the MR implication an increase in the filtering behaviour can be identified with an increase of the dof. However, differently from the dnl, the dof applies to each main frequency-amplitude component of the test. Hence we need to evaluate it for each component of the inputs, i.e., $\forall f | (f, A) \in fAmap[r_i(t)]$ and compare it with the corresponding component of the other test, i.e., the one with frequency $f\frac{\gamma_j}{\gamma_i}$ (leveraging the linearity of the DFT). Using this quantification in the implication of increasing dof we obtain

$$\textbf{MR2:}\ \begin{array}{c}(\phi_i = \phi_j) \wedge (\gamma_i > \gamma_j) \wedge (dnl(i) < dnl_{th}) \wedge (dnl(j) < dnl_{th}) \implies \\ \forall f | (f, A) \in fAmap[r_i(t)], dof(i, f) > dof(j, f\gamma_j/\gamma_i)\,.\end{array} \tag{5}$$

Property **PR3** states that, as long as the tests show linear behaviour, the closed-loop bandwidth should not depend on the specific test. To complement the two previous MRs that concern tests belonging to the same shape, we use this property to define an MR across different shapes. We identify the closed-loop bandwidth estimated with the tests of a given shape ϕ as the threshold $f_{b,\phi}$. As discussed in Section 4.2 the f_b can be evaluated for each shape as the frequency below which the dof of the different frequency components (of tests that show linear behaviour) is smaller than 0.5. Therefore the third MR can be defined the expectation that the f_b estimated using tests from different shapes are similar

$$\textbf{MR3:}\ |f_{b,\phi_i} - f_{b,\phi_j}| < \varepsilon, \tag{6}$$

where ε is a small discrepancy that can be accepted.

5.2 Approach Steps

In this section we describe the main steps of our testing approach. We use our test case parametrisation to generate test cases that cover different ranges of amplitudes

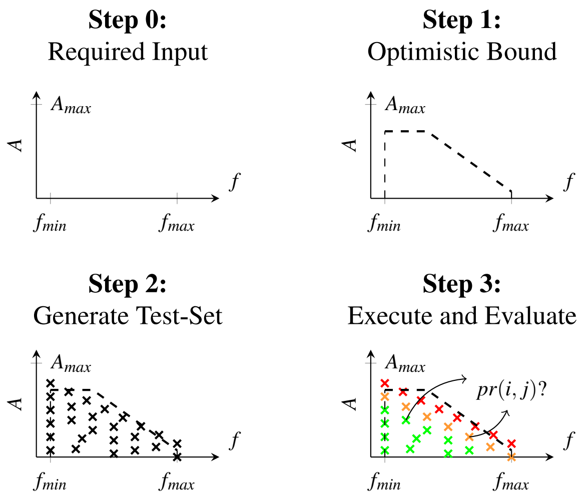


Figure 9. Graphical representation of main steps of proposed testing approach. For each step we describe the contribution to the testing in terms of the information that it adds in the frequency-amplitude plane. In each plot we show the bounds provided by the engineers as required input. The dashed line represents the optimistic bound on the amplitude values obtained in the Step 1. The crosses represent how the tests generated in Step 2 sample the frequency-amplitude plane. In Step 3 the colouring of the crosses and $pr(i, j)$? respectively exemplify the evaluation of the behaviour observed in the tests and the evaluation of the metamorphic relations.

and frequencies. Afterwards, we use the MRs enabled by the parametrisation to sanity check the results and the *dnl* to identify stress test cases.

In our approach, we require the engineers to provide ranges of frequency and amplitude values that are relevant for the SUT. Starting from these ranges, we propose a testing approach based on three main steps. We use Figure 9 to provide an approach overview and a graphical representation of how the required input and steps contribute to the generation and identification of stress test cases:

Step 0. **Required Input.** In this preliminary phase we ask the engineers to provide bounds on the relevant frequency and amplitude ranges as well as the desired resolution. We use the ranges (f_{min}, f_{max}, and A_{max} in Figure 9) to limit the scope of the testing to practically relevant values. The resolution is instead used to define how small variations in the reference value are expected to impact the control behaviour.

Step 1. **Optimistic Bounding of Amplitude Values.** This iterative step provides an optimistic evaluation of amplitude values that cause non-linear behaviour in the SUT (the dashed line in Figure 9). To obtain such values, we use tests with sinusoidal shape. The sinusoidal shape maps to an individual frequency-amplitude component and does not account for the combination of different components, hence giving its optimistic nature to this step.

Step 2. **Generate Test-Set.** This step uses different input shapes to generate the actual test-set. We use the optimistic amplitude bounds to limit the target area of the input space during the test-case generation. For each shape, we use uniform sampling of frequencies and random sampling over amplitudes to cover the target area (each of the black crosses in Figure 9 corresponds to a pair of frequency-amplitude coordinates, hence a generated test).

Step 3. **Tests Execution and Evaluation.** In this step we execute the tests and quantify the observed behaviours (the colouring of the crosses in Figure 9 symbolically represents the different behaviours). We sanity check the measured behaviours with the MRs (the $pr(t_i, t_j)$ in Figure 9). We use the degree of non-linearity to identify test cases that push the control algorithm out of its design scope to various degrees.

We now delve in to the details of each step.

Required Input: In order to practically initialise the approach we ask the engineers to define different quantities specific to the control system. Such quantities initialise the ranges of the amplitudes and frequencies that are relevant for the SUT, and define the desired resolution across the tests. The required quantities are:

- A bound on the maximum amplitude A_{max} that can be fed to the system[11]: this is an upper boundary to limit in practice the exploration of high values along the amplitude axis. For example, for drone altitude control, it can be set to the maximum altitude that the drone is expected to fly at.
- A frequency range $[f_{min}, f_{max}]$ around the expected closed-loop bandwidth f_b. Such expected value can be obtained from the control design process or from the speed requirements. As discussed above, it is important to cover the closed-loop bandwidth in order to push the CPS to its performance limits. As a rule of thumb, a factor of 10 around the expected f_b should be sufficient to make sure to include the actual one.[12]
- An amplitude resolution ΔA that is used to define how small variations in the input size are expected to have an impact on the system behaviour. Ideally, this should be a range within which differences in performance are expected

[11] The lower boundary is always zero since the amplitude describes the absolute value of the signal.

[12] With such rule of thumb, if the tests show that the f_b is not included in the used frequency range, it implies that there is an error of more than one order of magnitude on the expected system bandwidth. Such large error in the expected value detects the presence of some issue in the system development.

to either not matter or be indistinguishable because of practical limitations (e.g. sensor resolution). However, such fine grained resolution is not practically needed (nor achievable) and larger values can be used.

As we discuss later, we propose an automated approach to retrieve a desired frequency resolution from the results of Step 1. In fact, it can be difficult to intuitively define a desired frequency resolution. In case of specific needs, the engineers can set the desired frequency resolution manually (e.g., to control the number of tests on the base of a given testing budget).

Concerning the approach itself, we require the definition of an upper bound dnl_{th} on the degree of non-linearity. This quantity defines a *dnl* value above which tests are considered to be outside of the design scope of the SUT. Being a bound on the *dnl*, it can be interpreted as the maximum accepted relative amplitude of new frequency components that we can accept in the CPS. Accordingly, it can be chosen in the range of the maximum relative accepted deviation from the reference tracking. For example, for drone altitude control, we might accept up to $0.15\,\mathrm{m}$ of deviations from the reference when hovering around $1\,\mathrm{m}$ and in presence of wind, hence set $dnl_{th} = 0.15$.[13] Furthermore, its value does not need to be strict. In fact, for what concerns the test case generation, it is used only for the preliminary optimistic bounding of the amplitude values (Step 1). Hence, a larger value can be selected without compromising the approach effectiveness. However, as for the ranges, a conservative choice for this parameter results in a larger test-set.

In general, the mentioned values can be defined according to the domain knowledge of the SUT and the requirements on the tracking of the reference. Furthermore, they can be chosen in a conservative way (i.e., large ranges and high resolution) at the cost of a higher number of tests. They might however require interaction with the domain experts for the given CPS application.

Optimistic Bounding of Amplitude Values: The purpose of this step is to obtain, for the different frequencies, an optimistic evaluation of the input amplitude values α that push the control algorithm out of its design scope. This optimistic evaluation allows us to restrict the sampling of the frequency-amplitude plane and avoid trivially large amplitudes that will expose non-linear behaviour (i.e., $dnl > dnl_{th}$). Accordingly, it helps targeting the area of the input space where non-linearities start to appear.

To obtain this optimistic bound, we use sinusoidal inputs, i.e., $\phi = \sin$. As noted in Section 2.2 (Figure 3), sinusoidal inputs sample a single point in the frequency-amplitude plane and therefore avoid potential interactions between different components. Since those tests do not account for said interactions, the test provides only an *optimistic* evaluation of the SUT behaviour for that frequency-amplitude combination. Said in other words, even if a sinusoidal test with a given value of α and

[13] Note that, being the *dnl* based on the DFT, it has a linear proportion with time-domain values. Hence relative amplitudes in the two domains are equivalent.

Algorithm 1 Optimistic Bounding of Amplitude Values

```
function OptimisticAmplitudeBound($f_{min}$, $f_{max}$, $\delta A$, $A_{max}$,$dnl_{th}$)
    $A_{bound} \leftarrow$ binary_search_dnl_th($f_{min}, \delta A, dnl_{th}$)
    upperbound_dnl ($f_{min}$) $\leftarrow A_{bound}$
    $A_{bound} \leftarrow$ binary_search_dnl_th($f_{max}, \delta A, dnl_{th}$)
    upperbound_dnl ($f_{max}$) $\leftarrow A_{bound}$
    while max_gap (upperbound_dnl) $> \delta A$ do
        $f \leftarrow$ sample (upperbound_dnl)
        $A_{bound} \leftarrow$ binary_search_dnl_th ($f, \delta A, dnl_{th}$)
        upperbound_dnl ($f$) $\leftarrow A_{bound}$
    end while
    return upperbound_dnl
end function
```

γ exposes linear behaviour, it could happen that other shapes paired with the same values, expose non-linear behaviour (i.e. $dnl > dnl_{th}$). On the other side, if a sinusoidal test shows non-linear behaviour, the use of a different shape (with the same values for α and γ), is unlikely to show linear behaviour. In fact, by changing to a more complex shape, we are adding new frequency components to one component that was already sufficient to push the control algorithm out of its design scope.

We show the pseudocode implementing this step in Algorithm 1. The algorithm takes as inputs the frequency range, the desired amplitude resolution, the upper bound on the maximum amplitude value, and the non-linearity upper bound dnl_{th}. It then uses tests with sinusoidal inputs to sample the frequency-amplitude plane and obtain a frequency-dependant bound upperbound_dnl(f) of amplitude values.

On the high level, our algorithm iteratively samples the frequency axis, i.e., it samples values of γ. For each sampled frequency, it performs a binary search to obtain the minimum value $A_{bound} = \alpha$ for which the SUT shows $dnl > dnl_{th}$.[14] Accordingly, the obtained minimum amplitude value is assigned to the threshold upperbound_dnl(f) $\leftarrow A_{bound}$.

The sampling of the frequency axis starts with the sampling of the minimum f_{min} and maximum frequency values f_{max} provided as input. It uses the obtained values to initialize the bound upperbound_dnl. Afterwards, the iteration mentioned above starts (the while loop in the pseudocode). This iteration continues as long as the gap between two threshold values of subsequent frequencies in the bound is larger than the desired resolution max_gap(upperbound_dnl) δA. More rigorously, it continues until for each sampled f_i it holds that upperbound_dnl(f_{i+1}) $-$ upperbound_dnl(f_i) $< \Delta A$. Accordingly, at each loop iteration it identifies a pair of frequencies for which the condition above does not hold and samples a new frequency between them ($f \leftarrow$ sample (upperbound_dnl)).

[14] We note that, in order to perform this binary search, the algorithm is assuming that MR1 holds: i.e. that the dnl increases for increasing values of α.

Generate Test-Set: In this step we generate the actual test set. Given our test case parametrisation, we need to define a set of shape functions and different amplitude and time scaling parameters. We now discuss each of them.

Concerning the shape functions, we propose to use a set of shapes inspired by common practice in control engineering. In control engineering, the most common inputs used to evaluate a control algorithm are the step (instantaneous change of the reference) and ramp (linear change in the reference) [Åstrom and Murray, 2008]. Accordingly, we propose the use of shape functions that resemble said inputs: i.e., square, sawtooth, triangular, and trapezoidal waves. If other patterns are available from use cases for the specific CPS application, engineers can expand this set.

For each of the chosen shapes, we generate a set of (γ, α) pairs. We aim at exploring the area of the input space delimited by the upper-bound threshold identified in Step 1. Since one shape corresponds to more than one frequency-amplitude (f, A) point, we take as reference the point associated with the largest amplitude (in relative terms), which we call the *main component* of a shape. Accordingly, we use the α and γ parameters to move the main component of each shape and obtain test cases that cover the area delimited by the upper-bound threshold.

For what concerns the frequency axis, in the general case, there is no reason to test a specific range more than others. Therefore, along this dimension, we aim at uniformly covering (with the main component of the shape) the frequency range $[f_{min}, f_{max}]$. Practically, we sample frequencies at equal intervals in the given range. Since it can be difficult to have an intuition for a desired frequency resolution, we propose to compute a desired resolution δf from the results of Step 1. Specifically, we suggest to use the average frequency gap (i.e., $avg_i\{f_{i+1} - f_i\}$) obtained in the amplitude upper-bound. The intuition is that said gaps were obtained by imposing a maximum difference δA in the amplitude threshold for which the SUT shows non-linear behaviour. Therefore they should resemble a frequency variation for which the *dnl* does not change significantly. In specific cases, the engineers can adapt the sampling of the frequency axis according to application-specific needs. For example, if a frequency range is known to be particularly relevant for the specific SUT, a biased random sampling can be applied.

Differently, along the amplitude dimension, we are generally more interested in exploring the area with large amplitudes. For the amplitude values we can leverage the upper bound obtained at Step 1 and avoid sampling large amplitudes that, already with a sinusoidal input, would provide a very high dnl. Practically, we limit the sampling range with the optimistic upper bound obtained in each sampled frequency. Within this range, we propose to sample different amplitudes α according to a beta distribution skewed toward the higher values.[15] In order to define the number of samples, we use to base it on the desired amplitude resolution δA. Practically, we compute the number of tests for each frequency f with upperbound_dnl$(f)/\delta A$ to obtain a number of samples compatible with the desired resolution. Similarly to

[15] A beta distribution is a version of the more classical exponential distribution with bounded support.

the frequency dimension, application-specific sampling strategies might be adopted if areas of particular interest are given.

Tests Execution and Properties Verification: Once the test cases are defined, in this final step we proceed to executing and evaluating them. The evaluation includes the verification of the MRs and the identification of the stress test cases. More specifically, we identify the stress test cases as the ones that show a degree of non-linearity in the range between 0 and dnl_{th}. In fact, as mentioned in Section 4.3 test with $dn = 0$ are tests clearly within the design scope while tests with $dnl > dnl_{th}$ are tests clearly outside of the design scope. We now provide guidelines for the execution of this step. In the experimental section, we showcase in detail this final step of visualisation and analysis for our case of studies.

For each executed test we store the output trace and compute, according to the definitions given in Section 4: (i) the set of relevant (f,A) points, (ii) the degree of non-linearity, and (iii) one degree of filtering per (f,A) point. Such information can then be analysed by leveraging plots based on the frequency-amplitude characterisation.

For analysing the *dnl* (and identifying stress test cases), the main component of the test can be plotted in a frequency-amplitude plane using the associated (f,A) coordinates.[16] The marker can be coloured with a gradient that represents the measured dnl. The engineers can then use the colours to identify regions of the input space where non-linear behaviour appears. Using the colouring, MR1 can now be checked. In fact, MR1 enforces a pattern similar to the one shown in Figure 5 (assuming that red corresponds to $dnl > dnl_{th}$, green to $dnl = 0$ and the colour gradients to the values in the range $[0, dnl_{th}]$). Test cases that are at the boundaries between the colours and that deviate from said pattern are the stress test cases. The inspection of said test cases is expected to provide information about the phenomena inside the CPS that can cause the control algorithm to fail.

For what concerns MR2 and MR3, we use them for sanity check of the test outcomes. We note that those latter MRs (i) concern only the tests that show linear behaviour, (ii) do not depend on the input amplitude, and (iii) discuss the dof. Accordingly, we first select only the tests that fulfil $dnl < dnl_{th}$, i.e., the tests that show linear behaviour. From the remaining tests, we extract the individual (f,A) components and associated dof value. We then plot, separately for each shape, the dof of each frequency-amplitude point as function of its frequency coordinate f. MR2 states that an increase in the frequency content corresponds to an increase in the degree of filtering. Accordingly, the proposed frequency-dof plot such plot should show an increasing trend. MR3 states that the f_b should not depend on the specific shape of the input. We recall that the f_b is defined as the frequency below which $dof < 0.5$ and above which $dof > 0.5$. Accordingly, the trend of the degree

[16] We remark that the *dnl* characterises the whole test, hence there is no added information in plotting all of the frequency-amplitude components. Conversely, the inclusion of all of them is likely to make the plot difficult to interpret.

of filtering values for the different shapes should cross the 0.5 value around the same frequency. If said properties are not verified by some frequency-amplitude components, the associated tests should be inspected for possible faults in the testing process. Furthermore, if it is not possible to identify the f_b (because all of the dof values are above or below 0.5), it implies that the frequency range $[f_{min}, f_{max}]$ does not include the actual closed-loop bandwidth. This means that only one between the tracking and filtering behaviour is being tested. This is apparently a fault in the testing process that requires re-adjustment of the $[f_{min}, f_{max}]$ range and repetition of the process.

6. Empirical Evaluation

In this section, we empirically evaluate the proposed testing approach. We aim at evaluating the approach effectiveness in generating and identifying test cases that violate the linearity assumptions made during the control algorithm design. Furthermore, we want to evaluate its ability to provide insights about the specific non-linear phenomena limiting the control algorithm design scope, as well as its general applicability to different CPSs. Accordingly, we define the following three research questions (RQ).

RQ1: The objective of our approach is to generate test inputs that lead the system close to the applicability boundaries of control theory. In our approach, the test case generation (Step 2) does not include iterations and directly generates test cases through random sampling. Therefore, we ask whether our approach is able to *directly generate test cases that cover the area around the applicability bounds of control theory*—where "directly" refers to the fact that, after Step 1, the test case generation does not involve an iterative approach.

RQ2: Our approach is based on the qualitative characterisation of the input space proposed in Figure 5. Given the qualitative nature of the characterisation we developed metamorphic relations that capture its relevant properties in our tests. Accordingly, we ask *if said properties actually apply and if their verification can be used to identify the non-linearities limiting the design scope of the SUT.*

RQ3: Our approach aims at being system-agnostic and applicable to different types of control-based CPSs. Practically, we mean that it should be able to identify applicability boundaries of control theory independently of the specific SUT. For this reasons, we ask whether our approach is *effective independently of the specific sources of the non-linear behaviour in the SUT.*

This section is structured as follows: we first present and motivate the case of studies. Afterwards we present the results of the testing campaign. We then use the results to answer our research questions: for each question we first present our answering methodology and then discuss it using the tests results. We conclude the this empirical evaluation with a general discussion on the results, on the threats to validity, and the code and data availability.

6.1 Systems Under Testing

To answer our research questions, we apply our testing approach to the simulation models of two CPSs. We aim at covering the two most common types of algorithms found in traditional control theory: PID control and state-feedback [Åstrom and Murray, 2008; Desborough and Miller, 2002]. Accordingly we chose a drone controlled with PIDs and a DC servo (a continuous current motor) controlled with state-feedback. We implement the drone model using Python[17] and the DC servo using Sumulink[18].

We base our drone model on the Crazyflie drone developed by Bitcraze[19] that is completely open-source. This gives us full access to the source code, as well as software and design documentation. We use this information to develop a detailed Python model of the physics (based on the models used at control-design time [Greiff, 2017]) and of the control software (based on the source code[20] and the design documentation [Greiff, 2017]). We choose this SUT as we can have complete knowledge about the system design and it allows us to evaluate our testing approach using the very same information available during the CPS development. The control layer of the drone includes three separate control loops for the three directions: among those we select the altitude control—i.e. the control along the vertical direction. We chose the altitude control as, differently from the other control loops, it allows for quantification of the non-linear phenomena ground truth (that we leverage in the answer to RQ1). Despite being a small drone (size of $6 \times 6\,\mathrm{cm}$), we note that the Crazyflie implements control algorithms that are used also in larger drone applications [Mueller et al., 2016].

The model of the DC servo is based on laboratory hardware used for educational purposes at Lund University. The control layer of a DC servo receives from the user a desired angular position of the motor axis (the input reference r). On the base of such information and of the motor axis position measurements (the output y) it generates a voltage signal to be fed to the motor in order to make it move (the actuation signal). Our simulation model executes the very same C code that is run on the microcontroller used in the physical implementation of the CPS. Whilst being based on a CPS used for educational purposes, the considered DC servo is analogous to electrical motors found in common industrial applications like the joint of a robot or the steering axis of an autonomous car. We chose this SUT because electrical motors can expose, depending on the specific application, different types of non-linearities that can affect the behaviour of the control algorithm. This allows us to develop different but still practically relevant versions of the DC servo where

[17] `https://www.python.org/`
[18] `https://se.mathworks.com/products/simulink.html`
[19] `https://www.bitcraze.io/`
[20] `https://github.com/bitcraze/crazyflie-firmware`

we manually inject the common non-linearities. We retrieve a list of non-linearities from the "Discontinuities" folder of the Simulink library:

- *Saturation* represents the limitation of variables in the system: it generally affects actuators (that have limited capacity) and sensors (that can have a limited range of values that can be read).
- *Quantization* is the effect of the finite precision in the analog-to-digital conversion of the sensors signals.
- *Pulse-Width-Modulation* is a common technique used for digital-to-analog conversion, it can distort the actuation signals.
- *Coulomb Friction*: friction is always found in physical models, it is however hard to model and non-linear and for this reasons often neglected or approximated.
- *Dead-Zone* and *Backlash* are always found in mechanical gearing as they are caused by the play between the different cogs.[21]

We add to the list a quadratic friction model as it is a very common source of non-linear behaviour in practice—e.g. aerodynamic drag in a car or drone.[22] Among said sources of non-linearity, we consider saturation, quantization, and pulse-width-modulation to be always present as they are part of the interface between the cyber and physical part of the system. Differently, we introduce one by one the quadratic friction, the coulomb friction, the dead-zone and the backlash. We obtain therefore the following five versions of the DC servo according to the included non-linearities:

DC1 saturation (on actuation and sensor), quantization, pulse-width-modulation.

DC2 saturation (on actuation and sensor), quantization, pulse-width-modulation, and Coulomb friction.

DC3 saturation (on actuation and sensor), quantization, pulse-width-modulation, and quadratic friction.

DC4 saturation (on actuation and sensor), quantization, pulse-width-modulation, and dead-zone.

DC5 saturation (on actuation and sensor), quantization, pulse-width-modulation, and backlash.

To summarise we apply our approach to six different SUTs: the altitude control of the drone and the five different versions of the DC servo.

[21] Among the blocks listed in the Discontinuities folder we excluded the "rate limiter" as it is just a special case of saturation, and the relay as it is a very simple and old control approach, used only for non-critical systems.

[22] Quadratic friction was possibly not included among the standard blocks as it is easily implemented with other blocks.

Table 1. Chosen values for the testing approach required input for the altitude control of the drone and for the DC servo.

SUT	f_{min}	f_{max}	A_{max}	ΔA
Crazyflie Drone	0.1 Hz	2 Hz	6 m	0.05 m
DC servo (all versions)	0.005 Hz	3 Hz	2π rad	0.015 rad

6.2 Testing Settings

In this subsection we present the settings needed for the application of our testing methodology to the chosen SUTs. We first report on the required input (defined in Section 5.2) and the manual evaluation of the number of input repetitions needed to evaluate the *dnl* (defined in Section 4.2).

Selection of Required Input: We summarise in Table 1 the values chosen for the required input of our testing approach. We chose those values on the base of domain-knowledge of the SUT. Concerning the frequency range, both systems should be able to track the reference within a time span of the order of seconds. Accordingly, we place the expected closed-loop bandwidth around 1 Hz. To give an intuition, this corresponds to taking around 2 s to track a unit change in the reference value: 1 m for the drone and 1 rad for the DC servo. For the drone we choose in the frequency range $[0.1\,\text{Hz}, 2\,\text{Hz}]$. For the DC servo, we do not have a clear estimate from the documentation. Hence, we chose a broader range $[0.005\,\text{Hz}, 3\,\text{Hz}]$.

Concerning the maximum amplitude, we reason on the largest inputs that we can expect in practice for our SUTs. The Crazyflie is expected to mostly fly in indoor environments. We chose a maximum amplitude in the reference variations of 6 m, corresponding to a large room.[23] The DC servo, is supposed to track desired angular positions. Hence, we limited the reference amplitude changes to one full rotation of the motor axis, 2π rad.

Concerning the amplitude resolution, we reason on which variations of the input amplitude can cause a change in the behaviour of the SUT. For the drone this is in the order of centimetres: for example, asking the drone to reach 1.01 m or 1 m should not show a (significantly) different behaviour. Accordingly, we chose $\delta A = 5\,\text{cm}$ for the Crazyflie. The DC servo is expected to react to reference changes in the order of degrees. Since such devices are expected to be rather precise we chose a range smaller than the desired precision. More precisely, we chose one $400th$ of a rotation for the DC servo (less than one degree) $\delta A = 0.015\,\text{rad}$.

Finally, for the implementation of the testing approach, we have to set the non-linear threshold. The dnl_{th} threshold is connected to the maximum relative deviation from the reference that we can accept. We take a conservative approach and accept up to 15% of relative deviation. This means that we identify tests with more than 15% deviation to be trivially out of scope and therefore not interesting in terms of

[23] For example, the Crazyflie has been used for performing automated inventorying of supermarkets [Greiff et al., 2021].

Table 2. *dnl* of the preliminary tests computed using different numbers of periods for each SUTs. For the Crazyflie we can see that the value stabilizes when using five periods, hence that is the number of periods that we use in the implementation of our approach. For each of the DC servo versions the value detects the non-linear behaviour after six repetitions, hence we use seven to allow for some margin.

Evaluation of the Required Number of Input Iterations for *dnl* Computation										
# Periods	1	2	3	4	5	6	7	8	9	10
Drone	0.12	0.26	0.36	0.40	0.39	0.41	0.47	0.54	0.56	0.54
DC1	0.05	0.10	0.09	0.09	0.08	0.50	0.45	0.42	0.37	0.30
DC2	0.05	0.10	0.09	0.08	0.07	0.54	0.49	0.47	0.43	0.36
DC3	0.01	0.02	0.03	0.01	0.02	0.25	0.24	0.24	0.25	0.24
DC4	0.05	0.10	0.09	0.09	0.07	0.49	0.45	0.44	0.41	0.35
DC5	0.05	0.11	0.10	0.09	0.07	0.49	0.45	0.42	0.38	0.32

stress testing. For example, in the drone case study, this means that we are interested in tests that deviate of up to 15 cm when the drone is expected to hover at 1 m of height. Accordingly, we set $dnl_{th} = 0.15$.

Setting of Number of Input Iterations: In order to apply our testing approach to the CPSs, we need to define the number of input repetitions needed to compute the *dnl* (Section 4.2). For the sake of efficiency, we want to select the smallest number of repetitions (i.e., shorter tests) that gives sufficient accuracy in the *dnl* computation. To evaluate this trade-off, we performed a preliminary experiment, in which we manually run a single long test with large amplitude and scaling coefficients such that the SUT exposes non-linear behaviour. We detect non-linear behaviour by visually inspecting that the output does not follow the input reference. We use 10 input repetitions and compute the *dnl* using different trace lengths that correspond to different numbers of input periods. We then evaluate after how many input periods the *dnl* converges to a value that detects the non-linear behaviour (i.e. it is higher than the chosen dnl_{th}). As this evaluation is different for the different SUT, we run it for each of them before implementing our testing approach.

In order to bring the SUT out of their design scope, we run tests with the maximum amplitude and frequencies defined by the required input (Table 1). For the Crazyflie, a sinusoidal shape proved sufficient to cause non-linear behaviour. Differently, the DC servo showed linear behaviour for such test with sinusoidal input. Hence we resorted to a more complex (in terms of frequency components) shape to bring the SUT out of its design scope. Specifically, a steps shape caused non-linear behaviour.

We report in Table 2 the *dnl* values obtained for each of the SUTs using different numbers of periods. For the Crazyflie, the *dnl* value exceeds the threshold already using two periods. We chose however to use 5 periods as the value increases until then. For the DC servo we observe a similar pattern in each version of the case of study. The *dnl* is below the threshold when using 5 or less periods and then suddenly increases to a higher (and constant) value above the threshold for 6 or more periods.

Table 3. Number of tests executed for each SUT in Step 1 (sinusoidal upper-bounding of the non-linear threshold) and 3 (execution of the test-set generated at Step 2).

Number of Executed Tests			
Approach Step	Step 1	Step 3	Total
Crazyflie Drone	112	1012	1124
DC1	12	5100	5112
DC2 (Coulomb Friction)	12	5010	5022
DC3 (Quadratic Friction)	12	5100	5112
DC4 (Dead-Zone)	12	5100	5112
DC5 (Backlash)	12	5100	5112

To allow for some margin we chose to use 7 periods in our tests.

6.3 Output of the Testing Campaign

We now present the results of applying our testing approach to the SUTs. We first discuss the number of executed tests. Afterwards, we use Figures from 10 to 13 to report the results of the testing process. For each SUT we report two sets of figures. The figures are directly related to the execution of Step 3 of the approach (Section 5.2).[24] Such figures report the the dnl and dof observed in the tests.

Number of Tests: In Table 3 we report the number of tests executed for each SUT. The table reports the number of tests executed respectively for Step 1 and Step 3 (Step 2 does not require the execution of test cases). We note that the difference between Crazyflie and DC servo in the test-set size is caused by two factors. First the DC servo has wider ranges and higher resolution selected in the required input. Second, the execution of Step 1 reduces the number of tests needed to cover the input space of the drone altitude control. We note that the execution of the first step for the DC servo required a much smaller number of tests compared to the Crazyflie. This is due to the fact that, within the frequency-amplitude range defined in the required input, almost none of the sinusoidal test cases showed non-linear behaviour (as noted also in the evaluation of the number of iterations needed for the dnl computation). For this reason the binary search converged quickly to a value close to A_{max}. Since this happened equivalently for the different versions of the DC servo, the obtained sizes for the different test-sets (corresponding to the number of tests executed in Step 3) are the same. The only exception among the versions of the DC servo is **DC2**, for which Step 1 excluded part of the amplitude values at high frequencies.

Figures on dnl: This set of figures reports the measured dnl for each test. Figure 10 reports the dnl results for the Crazyflie, and Figure 12 reports the dnl results for

[24] In this subsection we only describe the figures. We analyse the validity of the MRs in the answers to the RQs.

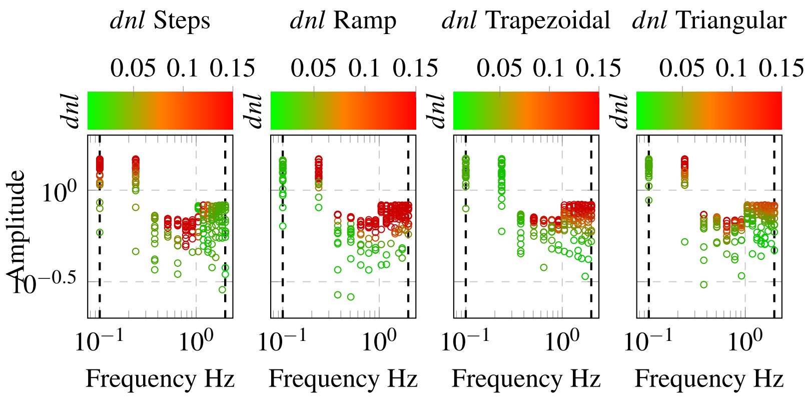


Figure 10. Degree of non-linearity of Crazyflie test cases divided by shapes. For each test the main component is plotted on the frequency-amplitude plane. Each point is coloured according to to the measured dnl: green corresponds to $dnl = 0$ and red corresponds to $dnl = dnl_{th}$ or greater.

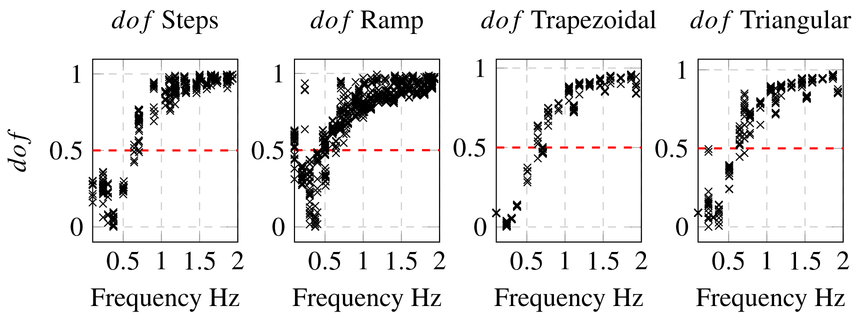


Figure 11. The figure reports the dof for all of the frequency-amplitude components for the tests that show linear behaviour (identified with $dnl < dnl_{th}$). The dof is reported as the vertical coordinate of each point and it is plotted over the frequency of the point (the horizontal coordinate). The plots are also used to identify the closed-loop bandwidth according to the tests made with different shapes: all of them cross the 0.5 threshold (highlighted by the red dashed line) giving a bandwidth $f_b \approx 0.6\,\text{Hz}$.

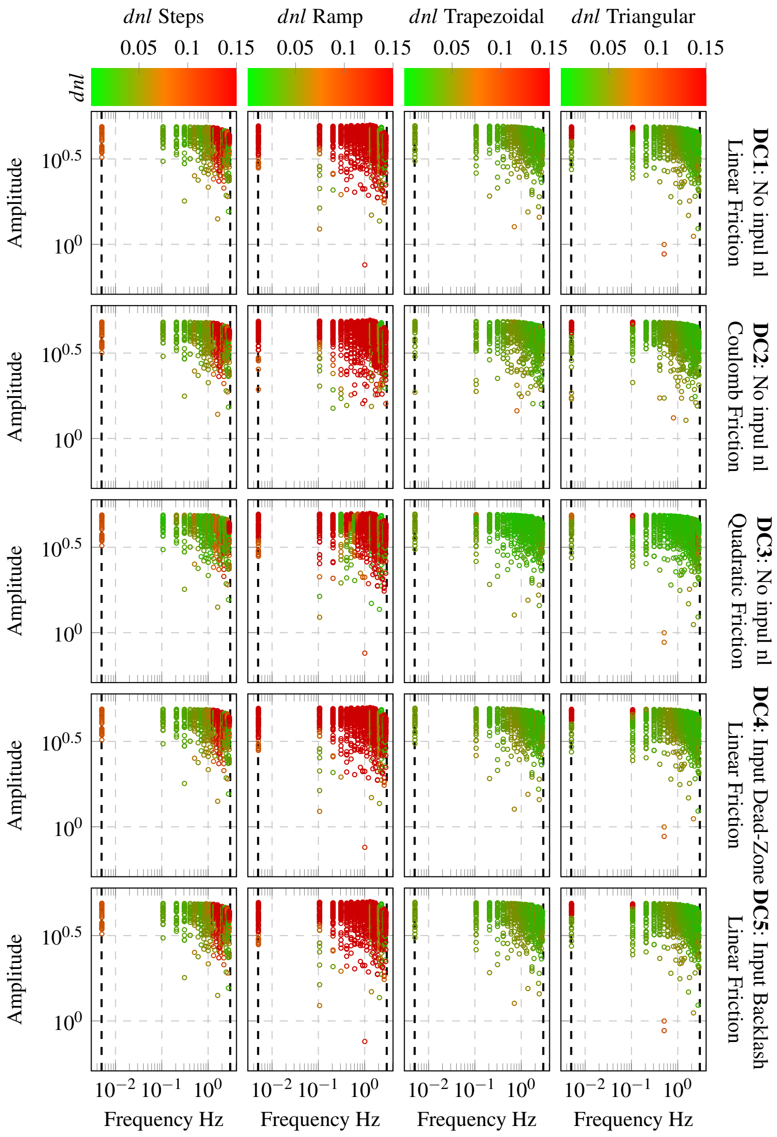


Figure 12. Evaluation of the *dnl* for the tests with different shapes (the columns) on the different versions of the DC servo (the rows). For each test we scatter plot the frequency-amplitude main component and colour it according to the measured *dnl*. The colour gradient has the same interpretation in every plot and goes from green that corresponds to $dnl = 0$ to red that corresponds to $dnl = dnl_{th}$.

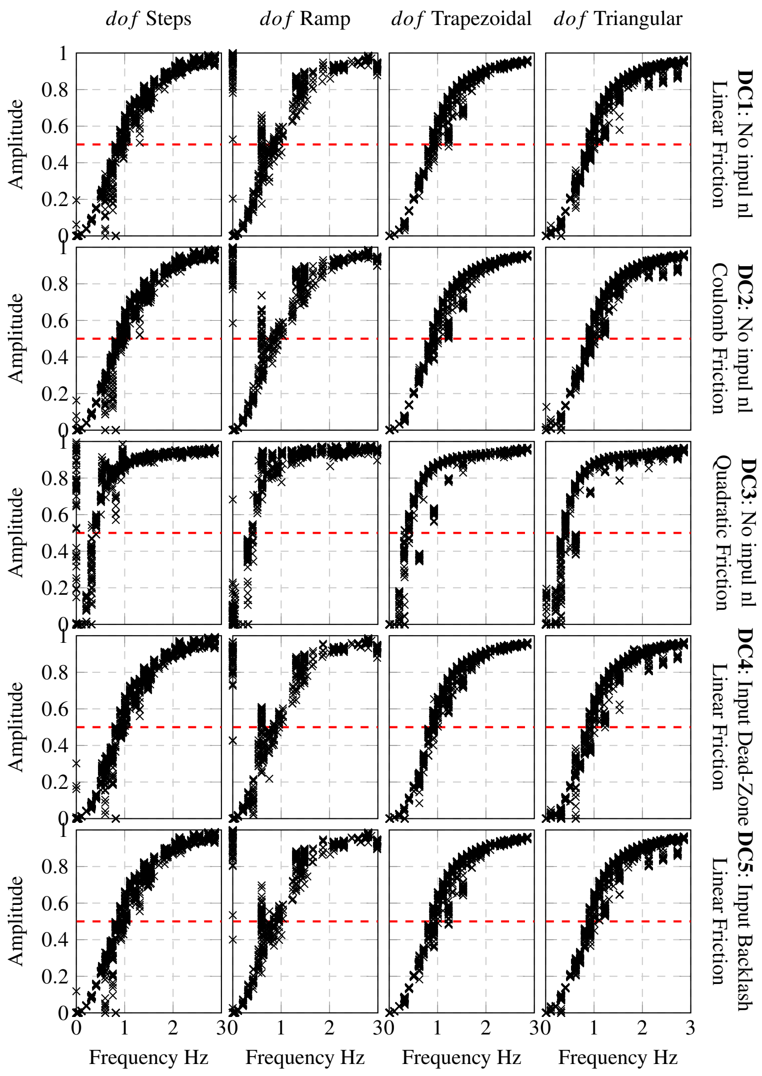


Figure 13. The figure reports the dof for all of the frequency-amplitude components for the tests that show linear behaviour (identified with $dnl < dnl_{th}$). The tests are separated in different plots according to the considered version of the DC servo (the rows) and the shape used in the test (the columns). In each plot the dof is reported as the vertical coordinate of each point and it is plotted over the frequency of the point (the horizontal coordinate).

the different DC servo versions (where the rows correspond to the different versions of the SUT). Since MR1 (i.e., the MR that concerns the *dnl*) discusses tests with the same shape, we separate these plots by shape. Specifically, each column reports the tests made with a given shape. As suggested in the approach description, we scatter plot the main frequency-amplitude component of each test on the frequency-amplitude plane. Like in Figure 7, we use logarithmic scale on both axes for readability. For each point, the colour gradient corresponds to the *dnl* measured in the test. Green corresponds to $dnl = 0$ (linear tests within the design scope) and red corresponds to $dnl \geq 0.15$ (non-linear tests outside of the design scope). Accordingly, the colour gradient between green and red highlights the stress tests.

Figures on dof: This set of figures reports the dof measured for all the frequency-amplitude points of the linear tests (i.e., the ones for which $dnl < dnl_{th}$). For the Crazyflie this is Figure 11, and for the DC servo this is Figure 12 (where again the rows correspond to the different versions of the SUT). Since both MR2 and MR3 (i.e., the MRs that concern the dof) discuss tests with the same shape, we separate these plots by shape (corresponding again to the different columns). Since, MR2 and MR3 do not concern the amplitude, we plot the dof (the vertical coordinate) of each point as function of its frequency only (the horizontal coordinate). Differently from the previous figure, we use here linear scale on both axes. In order to ease the identification of the closed-loop bandwidth f_b, we highlight with a red dashed line the 0.5 threshold for the dof.

6.4 RQ1 – Test Generation Effectiveness

Answering Methodology: To answer this question, we evaluate the ground truth occurrence of the different non-linear phenomena in the performed tests. If the testing approach is effective, it will generate test cases where non-linear phenomena appear to various degrees and affect the system performance.

Results: In our case of studies, to identify the non-linear phenomena ground truth, we can leverage our detailed knowledge of the SUTs, and the fact that we manually introduce (for the DC servo) the non-linear components. We now discuss, when possible, how we use such knowledge to quantify the non-linear phenomena in our SUTs.

In the case of the Crazyflie altitude control, we identify as non-linear phenomena the saturation of the actuators (i.e. the electric motors have a constrained power range) and the fact that the motors cannot generate negative force (the propellers cannot generate a force that pulls the drone down, hence downward movement is achieved only through gravity force). Both phenomena can be detected when the voltage signal sent to to the motors reaches its upper or lower limit. Accordingly, for a given test, we quantify the occurrence of non-linear phenomena as the percentage of time during the test that the actuators are saturated.

In the case of the different DC servo, we identify as non-linear phenomena the saturation (of both actuators and sensors), the pulse-width-modulation, the quan-

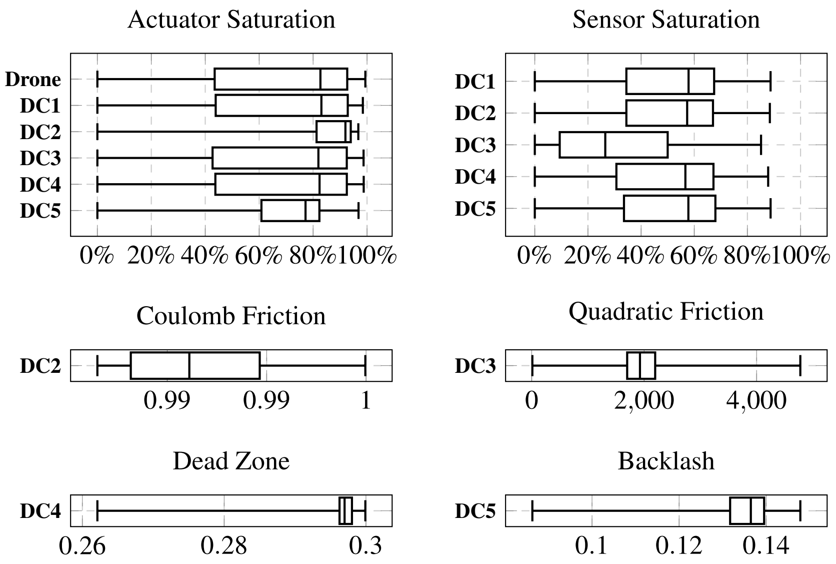


Figure 14. This figure reports the occurrence of the different non-linear phenomena in our tests. For each phenomenon we report on all of the SUTs that apply. Because of the different nature of the phenomena they have different ranges of values. However, we are only interested in the definition of a value that is zero when the model is behaving linearly and that increases when the behaviour differs further from the linear model. We are not interested in the comparison between the different phenomena.

tisation and the injected non-linearities. For what concerns the non-linearities that are always included, we can quantify the saturation (of both actuators and sensors) in the same way as for the Crazyflie. Differently, quantization and pulse-width-modulation affect every sensor reading and actuation in the control loop. Hence it is not possible to distinguish tests in which they appear more than others. Concerning instead the non-linearities injected in separate instances of the SUT, we quantify them as the deviation from how the model would have behaved in their absence. For example, when we inject the non-linear models of the friction, we can compare them to the friction value we would have obtained with a linear model. Accordingly, in every time instant we measure the difference between the two values. We then average such values over the whole test and obtain a metric for how much the non-linearity has appeared over the whole test.[25]

[25] We note that this approach can give very different value ranges according to the specific phenomenon considered. For example the input variation caused by the play found between cogs, i.e. the backlash or the dead-zone, will be much smaller than the friction variation caused by the use of a quadratic model instead of linear. However, for what concerns the answer to RQ1, we are only interested in

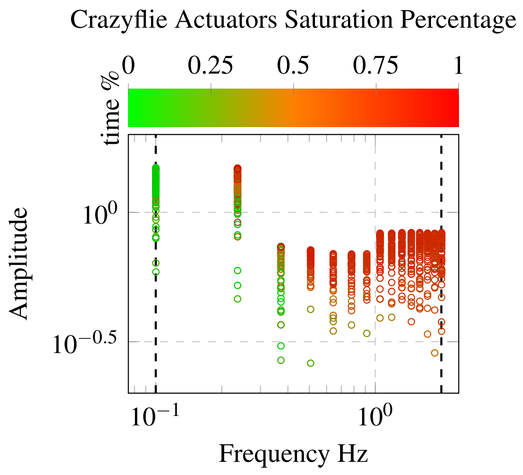


Figure 15. This figure reports the ground-truth of the actuator saturation in the Crazyflie altitude control tests. The figure shows, on the frequency-amplitude plane, one point for each test corresponding to the main component of the input sequence. For each point, the colour gradient shows the ground truth occurrence of actuator saturation: green corresponds to no actuator saturation (0% of test time) and red corresponds to complete saturation (100% of test time).

We report the measured ground truth for the different non-linear phenomena occurrence in Figure 14. The plots use the box plot convention to highlight the minimum and maximum values as well as the quartiles of the appearance of the non-linear phenomena over the different tests. We complement this quantification showing the same data in the frequency-amplitude plane. Analogously to the *dnl* plots we use the frequency-amplitude coordinates of the main component of the test and the colour gradient for the non-linearities. For the Crazyflie we have the actuator saturation in Figure 15. For the different DC servos, in Figure 16 the different columns correspond to the non-linear phenomena (the actuator and sensor saturations, and the injected non-linearity). Since the third column corresponds to the injected non-linearity, the colour gradient has different definitions in the different rows. In any case, in each plot, green corresponds to absence of the non-linear phenomenon and red to high presence.

Using said figures we now discuss the ability of the approach to expose the different non-linear phenomena. For what concerns the *actuator saturation*, the top left box plot in Figure 14 shows that our tests cover the full range of saturation percentages for each of our SUTs. This shows that our testing approach is able to generate test cases that trigger the saturation of actuators to any degree. Equivalently, Fig-

the definition of a value that is zero when the model is behaving linearly and that increases when the behaviour differs further from the linear model, hence there is no need to normalise the data.

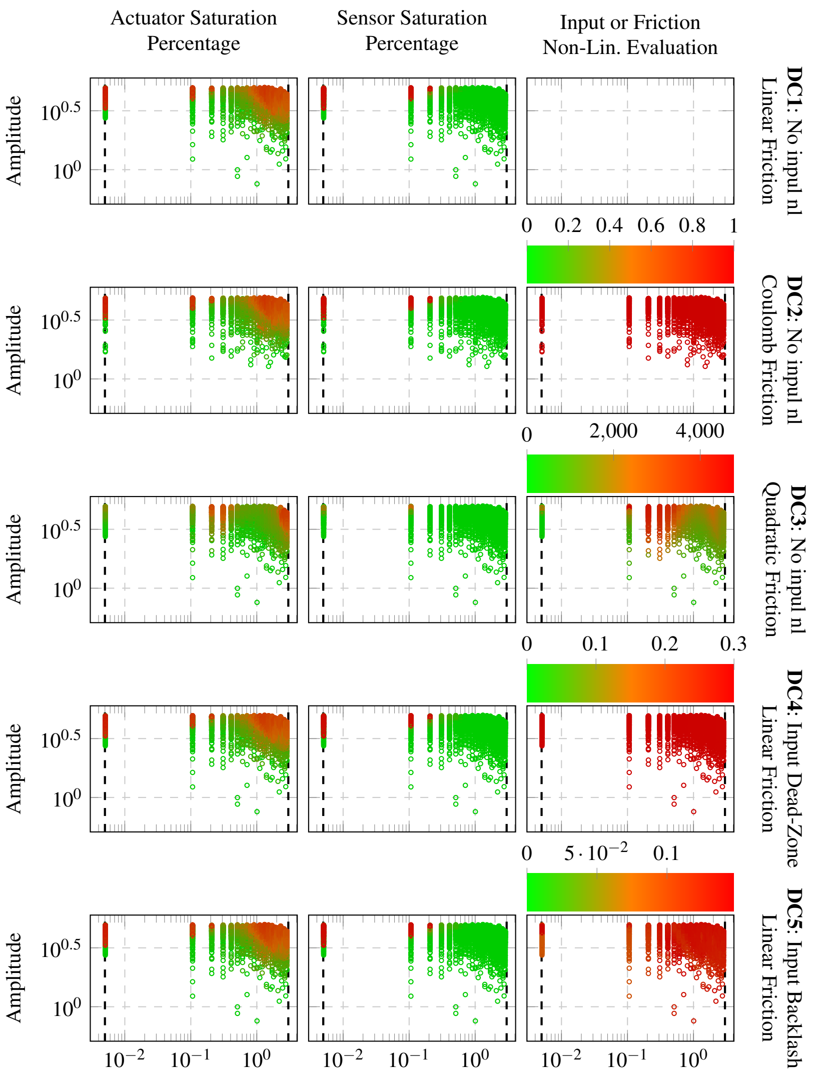


Figure 16. Evaluation of the non-linear phenomena ground truth for the different DC servo. The columns correspond to the non-linearities: the first two show the actuator and sensor saturation time percentages (green is 0% and red is 100%). The rightmost column shows the detection of the injected non-linearity in the friction (second and third rows) or in the actuation (fourth and fifth rows). The first row corresponds to the DC servo without injected non-linearity, hence the plot is empty. The colour gradient represents the detection of non-linear behaviour: green corresponds to zero values and red to the higher ones.

ure 15 for the Crazyflie and the left-most column of Figure 16 for the DC servo show that our tests cover the full scale of the colour gradient. Furthermore, from the latter plots, we can highlight that, by covering different amplitudes and frequencies, the approach is covering the transition areas of the frequency-amplitude plot where the actuator saturations start to appear.

When looking at the *sensor saturation*, which concerns only the DC servos, the top-right box plot in Figure 14 shows that our tests almost cover the full range of saturation percentages for each DC servo version. Our tests miss only values above 90%. The central column of Figure 16 shows that said tests appear for the lower frequencies and higher amplitudes. This is a consequence of the choice of A_{max} which is equal to the sensor saturation value. Accordingly, only the input sequences with the highest amplitude values can push the sensor to reach its maximum value. Such values are reached only for lower frequencies, while at higher frequencies the filtering action (which reduces the amount of input signal that is found in the output) prevents the saturation from happening. Therefore, thanks to the frequency coverage, we can note that our approach is still effective at triggering the saturation of the sensors, despite the limiting choice of A_{max}.

Lastly, we look at the other non-linear phenomena that we inject in the different versions of the DC servo, shown in the lower box plots in Figure 14 and in the right-most column of Figure 16. We observe that the coulomb friction and the input dead-zone are present in all tests to a high degree.This is to be expected since said non-linearities affect every movement in the DC servo and will therefore always appear. Differently, the quadratic friction is successfully exposed to a variety of degrees (Figure 14). Furthermore, in Figure 16 we can see that this variety is achieved mostly by the coverage of the frequency axis (since the colour gradient changes mostly when moving horizontally rather than vertically). Intuitively, friction is a phenomenon associated to the rate of change of the system, which is associated to the frequency axis.[26] Finally, the backlash appears in most tests exception made for the lower frequencies (Figure 16). In fact, the backlash affects the changes of direction of the input: apparently, for slowly changing signals (low frequencies), there are fewer changes of direction.

To summarise, our experiments show that the tests generated by our testing approach effectively trigger the non-linearities present in the SUT. The saturation and quadratic friction non-linearities are exposed to various degrees. Coulomb Friction, dead-zone, and backlash are instead easily involved in every execution of the SUT. By leveraging the proposed frequency-amplitude characterisation of the input space and the test case parametrisation, we observe that the test case generation can be made with random sampling and achieves good coverage of the non-linear phe-

[26] The quadratic friction is exposed the most around 0.2Hz, and 0.3Hz which from Figure13 we can see it is the closed-loop bandwidth of the system: at this frequency the system is behaving the "fastest" which causes the friction to be the largest and hence the difference between quadratic and linear is maximised (we recall that friction is function of the speed of motion). Beyond the f_b, the input is filtered and the non-linearity is exposed less.

nomena also without an iterative (search) approach.

6.5 RQ2 – Relevance of Metamorphic Relations

Answering Methodology: We discuss this RQ separately for the different MRs. For each MR we visually inspect the output of our testing process (Figures from 10 to 13). We then identify tests for which the MRs do not apply and discuss the causes. For the MRs to be practically relevant, the test cases that do not fulfil them should highlight phenomena that limit the SUT design scope.

MR1 states that the *dnl* should increase when moving to the right or up in the frequency-amplitude plane. Practically, this corresponds to the colour gradient transitioning from green to red for increasing amplitudes and frequencies in Figure 10 for the Crazyflie, and Figure 12 for the different DC servo. MR2 states that higher frequency content should correspond to higher degree of filtering. Practically, this corresponds to an increasing dof for increasing frequencies in Figures 11 for the Crazyflie and Figure 13 for the different DC servo. In the same figures we can also evaluate MR3. To fulfil MR3, the frequency at which the dof becomes larger than 0.5 (i.e., it crosses the red dashed threshold) has to be similar across the different shapes.

Results – MR1: For the Crazyflie (Figure 10), we observe that MR1 seems to hold for lower frequencies over the different shapes. However, for frequencies higher than 1Hz, in the tests with Steps and Triangular shapes, we observe more tests showing linear behaviour (i.e. green points). This happens despite our non-linear phenomena ground-truth in Figure 15 shows that the actuators are fully saturated in that frequency range. This is due to the filtering action that mitigates the response of the control-loop and allows it to retain non-linear behaviour and preventing the non-linearity from introducing new frequency content in the output. This statement can be further verified by observing in Figure 11 that the inputs in the frequency range above 1Hz show all a dof close to 1.

Concerning the DC servo (Figure 12), we observe a similar pattern as the one for the Crazyflie for the tests based on the Steps shape (as well as some of the high-frequency ramp tests). We then note that across the shapes, exception made for the trapezoidal tests, also tests at the lower frequencies and higher amplitudes (top-left of the plots) expose higher *dnl*, and do not comply with MR1. By comparing with the ground truth of Figure 16, we can observe that this is due to the combination of saturation of the sensor values and the actuator (first and second column).

Among the DC servo tests, it is notable that almost all of the ramp tests show non-linear behaviour, despite the ground truth (Figure 16) not detecting a specific non-linear phenomenon. The likely reason for this discrepancy with the other shapes is that the ramp is the shape with the highest number of main frequency components. By manually inspecting the individual tests we observe that they do show an undesired behaviour but we could not root cause it.[27] However, we recall that in the system there are other non-linear phenomena that could not be quantified (pulse-width-modulation and quantisation). Hence we can suppose that such tests fail because of said phenomena, or a combination of the different phenomena. On the other hand, we note that for lower amplitudes and frequencies some ramp tests expose a lower *dnl*. This still supports a partial validity of MR1.

To conclude, our tests results show that MR1 does not always hold. Especially for higher frequencies, the filtering behaviour can increase the robustness of the control algorithm to non-linearities. This remarks the qualitative nature of the proposed characterisation and specifically of Figure 5.

Results – MR2: In order to verify if MR2 applies to our tests results, we look at Figures 11 for the Crazyflie and 13 for the DC servo. We observe that in general the dof shows an increasing value for increasing frequencies, hence fulfilling the MR. The exceptions are some ramp tests that show an high dof for low frequency (with respect to the general behaviour of the tests). This happens for both the Crazyflie and the DC servo. Furthermore, also some steps tests on the DC servo injected with quadratic friction show a similar anomaly. Upon manual inspection, we observe that those tests are characterised by large inputs that cause saturations (of both sensors and actuators) to occur. However, in those tests, the output does not follow the reference (hence the high dof) but neither exposes new frequency components. For this reason, those test do not have a high *dnl* and were not excluded by the analysis. This exposes a *limitation of the dnl metric* to highlighting test that push the system outside of the control design scope. However, our results show that *the verification of MR2 serves as sanity check to compensate for this limitation of the degree of non-linearity*. In fact, the verification of MR2 highlights test cases in which the low frequency components (i.e., frequencies below the bandwidth) are not tracked.

Results – MR3: To evaluate MR3 we look again at Figures 11 for the Crazyflie and 13 for the DC servo. We observe that, for each of the SUTs, the crossing of the 0.5 threshold happens around the same frequency independently of the specific input shape considered. This suggests that MR3 generally holds in control-based CPSs and that it can be used to sanity check the results of the testing process.

To summarize, we observe that MR1 does not always hold but it enables the detection of test cases that are associated to the appearance of specific non-linear phenomena that characterise the SUT. MR2 holds in the majority of our tests. When it does not hold, our tests show that it can be used to complement the *dnl* metric to detecting test cases that push the system outside of the design scope. Finally, MR3 seems to be the one that holds more consistently across SUTs and input shapes. Therefore is can be used as a sanity check of the testing process.

[27] Root causing is outside of the scope of this work.

6.6 RQ3 – Generality to Different Non-Linearities

Answering Methodology: To answer RQ3, we assess if the different non-linear phenomena alter the effectiveness of the approach (RQ1) and the validity of the MRs (RQ2). Leveraging the different non-linearities that characterise our SUTs, we evaluate how the answers to RQ1 and RQ2 change across them.

Results: Concerning the first RQ, for all our six case of studies the approach has been able to generate test cases that expose the specific non-linearities (Figure 14). More specifically, saturations and quadratic friction are exposed to various degrees, while the remaining non-linearities are always exposed. The sensor saturation was detected in only a smaller number of test cases (for all the versions of the DC servo). However, this is possibly a consequence of the choice of the input parameter A_{max}. In fact, this bound on the amplitude is equal to the sensor saturation value, and the generated test inputs do not go beyond it.[28] The input and friction non-linearities are instead easily triggered in every test. However, not being associated with a high *dnl*, they do not impair the design scope of the control algorithm.[29]

Concerning the second RQ, the considerations are different for the different MRs. MR1 shows different validity and gives different types of insights across the SUTs. For the actuator saturation of the Crazyflie, MR1 is is violated because of the robustness introduced by the filtering action. Such robustness appears also in the DC servo case of studies, although to a smaller degree. For the DC servo, the sensor saturation invalidates the MR: in fact *dnl* increases for lower frequencies (e.g., going from 0.1 Hz to 0.01 Hz in Figure 12). Differently, the verification of both MR2 (as complement to the *dnl*) and MR3 (as sanity check) does not show significant differences across each of our SUTs. Most notably, the ramp tests that do not satisfy MR2 appeared similarly across all of the SUTs.

To summarise, in our case of studies, the stress test case generation effectiveness (RQ1) and the validity of MR2 and MR3 do not show significant dependence on the specific non-linearity. The verification of the MR1 instead, shows some differences related to the specific design bounds of the different SUTs. Specifically, it highlights the high-frequency robustness to saturations of the Crazyflie and the issues with sensor saturation for low frequency in the DC servo. This is desirable as it helps identifying the relevant phenomena for the given SUT.

6.7 Discussion

Our experiments show that the proposed testing approach is able to generate stress test cases for individual control-loops of control-based CPS. More specifically, how they falsify the design assumptions associated with the linearisation of the physics

[28] Testing with values that explicitly go beyond what the sensor saturation would belong to the domain of robustness testing (intended as testing with injected faults in the inputs) rather than stress testing.

[29] Such non-linearities (coulomb friction, dead-zone and backlash) can still impair other types of system performance, e.g., the accuracy of the reference tracking.

models (RQ1). Furthermore, we observed effectiveness independently of the specific non-linear phenomena that characterise the system (RQ3).

In practical scenarios, the non-linear phenomena ground truth is generally not available (i.e., Figures 15 and 16). For our tests cases, we showed that the *dnl* and MR1 can be used to highlight the test cases where specific non-linear phenomena limit the design scope of the control algorithm (RQ2-MR1). On the other hand, our tests also showed a limitation of the *dnl*. When the non-linear phenomena do not introduce new frequency components, the *dnl* can still take small values. Such tests are however detected using MR2 (RQ2-MR2). In fact this MR is invalidated (and hence highlights) the test cases where the filtering behaviour appears at low frequencies (where we would expect instead tracking).

Concerning the test case generation, we can note that the proposed rules of thumb for the approach required input generate a high number of test cases (Table 3). This can be a limiting factor when considering that we are testing only one control-loop of a CPS. In practice, not as many tests are needed and similar considerations can be made on the SUT with fewer frequency-amplitude points in Figures from 10 to 13. We leave to future work the challenge of minimising the number of test cases needed to stress test a control-loop. The problem of minimising the test cases can be in fact seen as a testing coverage or stopping criterion problem in the frequency-amplitude plane. In this work we focus on the test-case-generation side of the problem.

6.8 Limitations and Threats to Validity

We discuss the limitations of our work in terms of external and internal validity. Concerning the external validity, we discuss the generalisability of our observations to other CPSs. While we used a total of six case of studies (and based them on the common non-linearities that are highlighted in Simulink), it can be argued that five of them are based on the same physical process. However, our case of studies cover the standard control algorithms that are found in the vase majority of CPS applications, namely, PID control and state feedback [Åstrom and Murray, 2008; Desborough and Miller, 2002]. Furthermore, they are based on common applications of CPS. Drones are nowadays a well-established and wide-spread application, and DC servos are similarly found in common applications like autonomous cars and robots.

Concerning the internal validity, we discuss limitations of our research methodology. It can be noted that we based significant part of the RQs discussion on our own understanding of what are the non-linear phenomena that affect our SUTs. However, we developed our simulation models from scratch on the base of the actual physical systems (both available to us). The development from scratch gives us confidence on our complete understanding of the behaviour systems. The access to the physical system allowed us to relate to the actual implementation of the CPS when uncertain on the model design choices (e.g., the specific quantisation of the sensors and the parameters of the control algorithms).

Finally, it can be noted that we used visual inspection to evaluate our MRs. While this allows only for a qualitative evaluation and not a quantitative one, it also enabled the intuitive leveraging of the frequency-amplitude plane representation. Such representation helped the association of the physical phenomena to the features of the input (e.g. the quadratic friction affecting more the fast-changing inputs). This enabled a deeper understanding of how the non-linear phenomena can limit the design scope of the CPS under test.

6.9 Data Availability

In two git repositories we provide the code of the SUT models and the code implementing the testing approach:

- **Crazyflie**: `https://doi.org/10.5281/zenodo.7274113`,
- **DC servo**: `https://doi.org/10.5281/zenodo.7274107`.

The repositories come with instructions to reproduce all of the experiments. Since the tests take some time to execute (around 3 days on a 2017 MacBook Pro, with 2.3 GHz Dual-Core Intel Core i5 processor), we provide the test output traces that can be used to obtain the figures of the paper.

- **Crazyflie**: `https://archive.control.lth.se/attic/claudio/cfdata_nlmax015.zip`,
- **DC servo**: `https://archive.control.lth.se/attic/claudio/dcServo_test_data_7periods.zip`.

7. Conclusion

In this paper we have defined the problem of stress testing control-based CPS. For CPS developed with the use of control theory, we have highlighted the different types of assumptions that the control engineers make during the design of the CPS control-layer software. Once the CPS is implemented, such assumptions are not always valid. Accordingly, they can limit the set of scenarios in which the SUT is able to fulfil its requirements. To help the engineers identify said scenarios, we have developed a testing approach that pushes the system at the validity boundaries of the linearity design assumptions (i.e. the design assumption needed by control engineers to use linearised versions of the physical models). We provided a qualitative characterisation of the CPS control-layer input-space and proposed a novel test-case parametrisation. Leveraging our test-case parametrisation we use the qualitative characterisation to generate and identify stress test cases, and to define metamorphic relations describing the expected CPS behaviour. We applied the proposed approach to six case studies and evaluated output of the testing process. Our results show that our testing approach effectively generates test cases that falsify the linearity design assumption and push the SUT at the bounds of its design scope. Furthermore, the metamorphic relations highlight relevant test cases for the definition of the CPS design scope and for sanity check of the testing process.

Future Work: One of the contributions of this work is to open a new perspective on the testing of control-based CPS, i.e., the stress testing driven by the control design assumptions. In future work we plan to extend this testing approach to the testing of other assumptions not considered here: more specifically, the neglecting of the control-mode changes and of the interaction between different control-loops. Furthermore, another important research direction is the testing of the combined falsification of the different assumptions. For example, exploring the testing of how the falsification of the linearised model combined with the falsification of the mode changes can affect the design scope of the SUT.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH). The responsibility for the content remains with the authors.

References

Abbas, H. and B. Bonakdarpour (2022). "Leveraging system dynamics in runtime verification of cyber-physical systems".

Afzal, W., R. Torkar, and R. Feldt (2009). "A systematic review of search-based testing for non-functional system properties". *Information and Software Technology* **51**:6, pp. 957–976. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2008.12.005`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584908001833`.

Aleti, A. and L. Grunske (2015). "Test data generation with a kalman filter-based adaptive genetic algorithm". *Journal of Systems and Software* **103**, pp. 343–352. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2014.11.035`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121214002660`.

Åstrom, K. J. and R. M. Murray (2008). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA. ISBN: 0691135762.

Ayerdi, J., V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel (2021). "Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE'2021. ACM, Athens, Greece, pp. 1264–1274. ISBN: 9781450385626. DOI: 10.1145/3468264.3473920. URL: https://doi.org/10.1145/3468264.3473920.

Balasubramaniam, B., H. Bagheri, S. Elbaum, and J. Bradley (2020). "Investigating controller evolution and divergence through mining and mutation*". In: *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 151–161. DOI: 10.1109/ICCPS48487.2020.00022.

Banerjee, A., S. Chattopadhyay, and A. Roychoudhury (2016). "Chapter three - on testing embedded software". In: Memon, A. (Ed.). Vol. 101. Advances in Computers. Elsevier, pp. 121–153. DOI: https://doi.org/10.1016/bs.adcom.2015.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0065245815000662.

Bradley, J. M. and H. Bagheri (2020). "Control software: research directions in the intersection of control theory and software engineering". In: *AIAA Scitech 2020 Forum*. DOI: 10.2514/6.2020-2102. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2020-2102. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2020-2102.

Briand, L., S. Nejati, M. Sabetzadeh, and D. 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, Austin, Texas, pp. 789–792. ISBN: 9781450342056. DOI: 10.1145/2889160.2889212. URL: https://doi.org/10.1145/2889160.2889212.

Bringmann, E. and A. 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, Shanghai, China, pp. 13–20. ISBN: 1595934022. DOI: 10.1145/1138474.1138479. URL: https://doi.org/10.1145/1138474.1138479.

Bringmann, E. and A. Krämer (2008). "Model-based testing of automotive systems". In: *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 485–493. DOI: 10.1109/ICST.2008.45.

Broy, M., I. H. Kruger, A. Pretschner, and C. Salzmann (2007). "Engineering automotive software". *Proceedings of the IEEE* **95**:2, pp. 356–373. DOI: 10.1109/JPROC.2006.888386.

Chen, T. Y., F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou (2018). "Metamorphic testing: a review of challenges and opportunities". *ACM*

Comput. Surv. **51**:1. ISSN: 0360-0300. DOI: 10.1145/3143561. URL: https://doi.org/10.1145/3143561.

Cooley, J. W. and J. W. Tukey (1965). "An algorithm for the machine calculation of complex fourier series". *Mathematics of Computation* **19**, pp. 297–301.

Desborough, L. and R. Miller (2002). "Increasing customer value of industrial control performance monitoring honeywell s experience". In:

Dreossi, T., T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh (2015). "Efficient guiding strategies for testing of temporal properties of hybrid systems". In: Havelund, K. et al. (Eds.). *NASA Formal Methods*. Springer International Publishing, Cham, pp. 127–142. ISBN: 978-3-319-17524-9.

Garousi, V., M. Felderer, Ç. M. Karapçak, and U. Ylmaz (2018). "Testing embedded software: a survey of the literature". *Information and Software Technology* **104**, pp. 14–45. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.06.016. URL: https://www.sciencedirect.com/science/article/pii/S0950584918301265.

Ghosh, B., C. Hobbs, S. Xu, P. S. Duggirala, J. H. Anderson, P. S. Thiagarajan, and S. Chakraborty (2022). "Statistical hypothesis testing of controller implementations under timing uncertainties". In: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 11–20. DOI: 10.1109/RTCSA55878.2022.00008.

Gille-Maisani, J. and P. Decaulne (1959). *Feedback Control Systems: Analysis, Synthesis, and Design*. McGraw-Hill series in control systems engineering. McGraw-Hill. URL: https://books.google.se/books?id=9WG9zQEACAAJ.

Greiff, M. (2017). *Modelling and control of the crazyflie quadrotor for aggressive and autonomous flight by optical flow driven state estimation*. eng. Student Paper.

Greiff, M., P. Persson, Z. Sun, K. Åström, and A. Robertsson (2021). *Quadrotor control on $SU(2) \times R^3$ with slam integration*. DOI: 10.48550/ARXIV.2110.01099. URL: https://arxiv.org/abs/2110.01099.

Hänsel, J., D. Rose, P. Herber, and S. 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*, pp. 170–179. DOI: 10.1109/ICST.2011.37.

He, X., X. Wang, J. Shi, and Y. Liu (2020). "Testing high performance numerical simulation programs: experience, lessons learned, and open issues". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Association for Computing Machinery, Virtual Event, USA, pp. 502–515. ISBN: 9781450380089. DOI: 10.1145/3395363.3397382. URL: https://doi-org.ludwig.lub.lu.se/10.1145/3395363.3397382.

He, Z., Y. Chen, E. Huang, Q. Wang, Y. Pei, and H. Yuan (2019). "A system identification based oracle for control-cps software fault localization". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 116–127. DOI: 10.1109/ICSE.2019.00029.

Khalil, H. (2002). *Nonlinear Systems*. Pearson Education. Prentice Hall. ISBN: 9780130673893. URL: https://books.google.se/books?id=t%5C_d1QgAACAAJ.

Lamberg, K., M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey (2004). "Model-based testing of embedded automotive software using mtest". In: *SAE 2004 World Congress and Exhibition*. SAE International. DOI: https://doi.org/10.4271/2004-01-1593. URL: https://doi.org/10.4271/2004-01-1593.

Lee, E. A. (2015). "The past, present and future of cyber-physical systems: a focus on models". *Sensors* **15**:3, pp. 4837–4869. ISSN: 1424-8220. DOI: 10.3390/s150304837. URL: https://www.mdpi.com/1424-8220/15/3/4837.

Magnani, G., D. Cattaneo, M. Chiari, and G. Agosta (2021). "The Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control". In: Bispo, J. et al. (Eds.). *12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021)*. Vol. 88. Open Access Series in Informatics (OASIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:13. ISBN: 978-3-95977-181-8. DOI: 10.4230/OASIcs.PARMA-DITAM.2021.3. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13639.

Marculescu, B., R. Feldt, R. Torkar, and S. Poulding (2015). "An initial industrial evaluation of interactive search-based testing for embedded software". *Applied Soft Computing* **29** (0), pp. 26–39. DOI: http://dx.doi.org/10.1016/j.asoc.2014.12.025. URL: http://www.sciencedirect.com/science/article/pii/S1568494614006693.

Matinnejad, R., S. Nejati, L. Briand, and T. 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, Vasteras, Sweden, pp. 163–174. ISBN: 9781450330138. DOI: 10.1145/2642937.2642978. URL: https://doi.org/10.1145/2642937.2642978.

Matinnejad, R., S. Nejati, and L. 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, Paderborn, Germany, pp. 938–

943. ISBN: 9781450351058. DOI: 10.1145/3106237.3117770. URL: https://doi.org/10.1145/3106237.3117770.

Menghi, C., S. Nejati, L. C. Briand, and Y. I. Parache (2019). "Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification". *CoRR* **abs/1910.02837**. arXiv: 1910.02837. URL: http://arxiv.org/abs/1910.02837.

Mueller, M. W., M. Hehn, and R. DAndrea (2016). "Covariance correction step for kalman filtering with an attitude". *Journal of Guidance, Control, and Dynamics*, pp. 1–7.

Nejati, S., K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. 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, Tallinn, Estonia, pp. 1015–1025. ISBN: 9781450355728. DOI: 10.1145/3338906.3340444. URL: https://doi.org/10.1145/3338906.3340444.

Peleska, J. (2002). "Hardware/software integration testing for the new airbus aircraft families." *http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html*. DOI: 10.1007/978-0-387-35497-2_24.

Plaku, E., L. E. Kavraki, and M. Y. Vardi (2009). "Falsification of ltl safety properties in hybrid systems". In: Kowalewski, S. et al. (Eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 368–382. ISBN: 978-3-642-00768-2.

Priyadarshi Tripathy, K. N. (2008). "Acceptance testing". In: *Software Testing and Quality Assurance*. John Wiley and Sons, Ltd. Chap. 14, pp. 450–470. ISBN: 9780470382844. DOI: https://doi.org/10.1002/9780470382844.ch14. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470382844.ch14. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470382844.ch14.

Samad, T. and G. Balas (2003). "Frontmatter". In: *Software-Enabled Control: Information Technology for Dynamical Systems*, pp. i–xx. DOI: 10.1002/047172288X.fmatter.

Sanchez-Stern, A., P. Panchekha, S. Lerner, and Z. Tatlock (2018). "Finding root causes of floating point error". *SIGPLAN Not.* **53**:4, pp. 256–269. ISSN: 0362-1340. DOI: 10.1145/3296979.3192411. URL: https://doi.org/10.1145/3296979.3192411.

Silano, G., E. Aucone, and L. 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)*, pp. 1–6. DOI: 10.1109/MED.2018.8442759.

Smirnov, S., J. Ania-Castanon, T. Ellingham, S. Kobtsev, S. Kukarin, and S. Turitsyn (2006). “Optical spectral broadening and supercontinuum generation in telecom applications”. *Optical Fiber Technology* **12**:2, pp. 122–147. ISSN: 1068-5200. DOI: `https://doi.org/10.1016/j.yofte.2005.07.004`. URL: `https://www.sciencedirect.com/science/article/pii/S1068520005000416`.

Timperley, C. S., 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)*, pp. 331–342. DOI: `10.1109/ICST.2018.00040`.

Vreman, N., A. Cervin, and M. Maggio (2021). “Stability and performance analysis of control systems subject to bursts of deadline misses”. English. In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Vol. 196. 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) ; Conference date: 05-07-2021 Through 09-07-2021. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: `10.4230/LIPIcs.ECRTS.2021.15`. URL: `https://www.ecrts.org/`.

White, A. (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. DOI: `10.1109/AERO.2001.931302`.

Wu, M., H. Zeng, C. Wang, and H. Yu (2017). “Invited: safety guard: runtime enforcement for safety-critical cyber-physical systems”. In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6. DOI: `10.1145/3061639.3072957`.

Yamagata, Y., S. Liu, T. Akazaki, Y. Duan, and J. Hao (2021). “Falsification of cyber-physical systems using deep reinforcement learning”. *IEEE Transactions on Software Engineering* **47**:12, pp. 2823–2840. DOI: `10.1109/TSE.2020.2969178`.

Yi, X., L. Chen, X. Mao, and T. Ji (2017). “Efficient global search for inputs triggering high floating-point inaccuracies”. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 11–20. DOI: `10.1109/APSEC.2017.7`.

Zander, J., I. Schieferdecker, and P. Mosterman (2011). *Model-Based Testing for Embedded Systems*. ISBN: 9781439818459.

Zimmer, M., J. Hedrick, and E. A. Lee (2015). “Ramifications of software implementation and deployment: a case study on yaw moment controller design”. *2015 American Control Conference (ACC)*, pp. 2014–2019.

Making Sure that your Car, and Many Other Things, are Safe

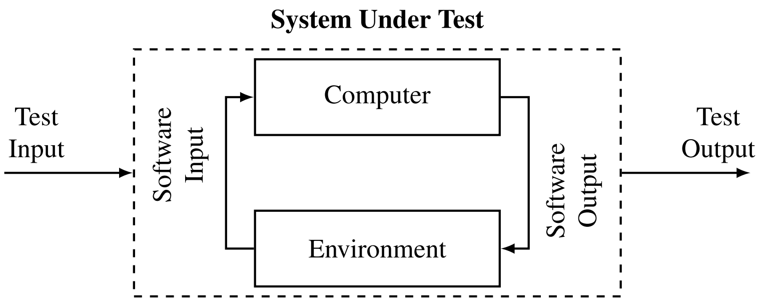
Claudio Mandrioli
Institutionen för Reglerteknik

Populärvetenskaplig sammanfattning av doktorsavhandlingen *Control-Theoretical Perspective on Feedback-Based Systems Testing*, december 2022. Avhandlingen kan laddas ner från: `http://www.control.lth.se/publications`

Computers are becoming smaller and more pervasive in the 21st century. Furthermore, we expect them to perform more and more critical tasks. For example, we expect the onboard computer of an aircraft to take care of most of the flight: take-off and landing included. Or we expect our car to drive itself: for now only in simple situations like the highway, but soon we will give it full control also in more complex situations like urban driving. We therefore find ourselves, on a daily basis, consciously or unconsciously, trusting a computer with our lives. But how do we really know that these computers will do the right thing every time?

The answer might seem surprisingly simplistic, but we just test them many many times! Once we have tested enough in a controlled environment, and we feel confident that the computer will be able to consistently operate in a safe way, then we give it the green light to head out in the chaotic outside world. However, this is apparently an imperfect process that does not always work successfully: take as an example the recent crashes of the Boeing 737 Max. It is therefore of prime importance that when we perform these tests we are as rigorous as possible so that we detect and remove all of the possible faults.

In this work we improved the methods and tools that we have available to test systems that are controlled by a computer. Meaning those systems in which we expect a computer to measure what is going on around itself and take quick executive decisions on what to do. The testing methods and tools are what we use to decide



The testing of a computer-managed system.

things like: how many times do we need to test the system? Which tests should we execute? Are the tests' results good or bad?

In this thesis we improved several aspects of this testing process. We improved how we can decide how many tests we need to run by formulating the question of the number of tests as a mathematical optimisation problem. We empirically investigated in which ways the environment in which we perform the testing can impact the effectiveness of the process. We developed a method to generate test cases that pushes the system to its performance limits, so that we can investigate what are the ranges in which the system can operate safely.

So, next time you will get on your car to go to the gym, or get on a plane for your next vacation, you can think of all the work that has been put, not only in making sure that it works, but also in making sure that it will always work. And hopefully, you can feel a little more confident that the computers taking decisions on how to drive the car or fly the plane have been thoroughly tested and they will make the right decisions for your safety.