Automated Testing of Robotic and Cyberphysical Systems

Afsoon Afzal

CMU-ISR-21-105

May 2021

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues (Chair)
Michael Hilton
Eunsuk Kang
John-Paul Ore (North Carolina State University)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This research was supported in part by the National Science Foundation (CCF-1563797, CCF-1446966), the Defense Advanced Research Projects Agency (FA-87501620042), and the Air Force Research Laboratory (GG-13332155259, GG11821148036).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: testing cyberphysical systems; robotics testing; automated quality assurance; simulation-based testing; challenges of testing; automated oracle inference; automated test generation;

To Pouneh, Arash, and all other 174 innocent victims of flight PS752.

Abstract

Robotics and cyberphysical systems are increasingly being deployed to settings where they are in frequent interaction with the public. Therefore, failures in these systems can be catastrophic by putting human lives in danger and causing extreme financial loss. Large-scale assessment of the quality of these systems before deployment can prevent these costly damages.

Because of the complexity and other special features of these systems, testing, and more specifically automated testing, faces challenges. In this dissertation, I study the unique challenges of testing robotics and cyberphysical systems, and propose an end-to-end automated testing pipeline to provide tools and methods that can help roboticists in large-scale, automated testing of their systems. My key insight is that we can use (low-fidelity) simulation to automatically test robotic and cyberphysical systems, and identify many potentially catastrophic failures in advance at low cost.

My core thesis is: Robotic and cyberphysical systems have unique features such as interacting with the physical world and integrating hardware and software components, which creates challenges for automated, large-scale testing approaches. An automated testing framework using software-in-the-loop (low-fidelity) simulation can facilitate automated testing for these systems. This framework can be offered using a clustering approach as an automated oracle, and an evolutionary-based automated test input generation with scenario coverage fitness functions.

To support this thesis, I conduct a number of qualitative, quantitative, and mixed method studies that 1) identify main challenges of testing robotic and cyberphysical systems, 2) show that low-fidelity simulation can be an effective approach in detecting bugs and errors with low cost, and 3) identify challenges of using simulators in automated testing.

Additionally, I propose automated techniques for creating oracles and generating test inputs to facilitate automated testing of robotic and cyberphysical systems. I present an approach to automatically generate oracles for cyberphysical systems using clustering, which can observe and identify common patterns of system behavior. These patterns can be used to distinguish erroneous behavior of the system and act as an oracle.

I evaluate the quality of test inputs for robotic systems with respect to their reliability, and effectiveness in revealing faults in the system. I observe a high rate of non-determinism among test executions that complicates test input generation and evaluation, and show that coverage-based metrics are generally poor indicators of test input quality. Finally, I present an evolutionary-based automated test generation approach with a fitness function that is based on scenario coverage. The automated oracle and automated test input generation approaches contribute to a fully-automated testing framework that can perform large-scale, automated testing on robotic and cyberphysical systems in simulation.

Acknowledgments

This dissertation would not be possible without the support and guidance from my advisor, my collaborators, my family, and my friends. First and foremost, I would like to thank my advisor Claire Le Goues, who has provided me with utmost support throughout my PhD. I could not have asked for a better advisor, who always provided me with the opportunity to follow my interests, and taught me the ups and downs of research. She has not only been a great advisor, researcher, and teacher, but also a great friend, supporting me through the tough times in my personal and professional life.

I would like to thank my committee members Michael Hilton, Eunsuk Kang, and John-Paul Ore. I greatly appreciate their time and feedback. Their input to my research and dissertation is invaluable. Special thanks to my closest collaborator and dear friend Christopher Timperley, for all the help and support he provided me over the years, from countless meetings and brainstorming sessions on various research projects, to terrific insights on video games and TV shows. I would like to thank all other collaborators Yuriy Brun, Kathryn Stolee, Deborah Katz, Manish Motwani, and Jeremy Lacomis, for their guidance and inspiration in research. It has been my pleasure to work with these wonderful researchers, and I wish we can extend our collaboration in the future.

None of my accomplishments would have been possible without the support, love, and care of my parents Farzaneh Adni-Azar and Mohsen Afzal. I am forever indebted to them for the opportunities they provided me to be able to follow my dreams. They have always been selflessly supporting me in my decisions, encouraging me to believe in myself, and sacrificing their own happiness for my success. Special thanks to my partner Ali Oğuz Polat for always being my partner in crime, and being there when I needed support. I am grateful that we shared our PhD journey together.

I would like to thank my friends Sareh Karami, Mehdi Ghahremani, and their precious baby Termeh, Sareh Yousefzadeh, Mostafa Mirshekari, Mahnoush Babaei, Mahbaneh Eshaghzadeh, Faezeh Movahedi, Fattaneh Jabbari, and Anahita Mohseni with whom I made many wonderful memories. Special thanks to my friends Negar Naghashzadeh and Hanieh Kazari for all the fun trips and adventures. I would like to also thank my friends and colleagues at Carnegie Mellon. Thanks to David Widder, Gabriel Ferreria, and Shurui Zhou for being wonderful friends. Thanks to all past and present members of the *SquaresLab* Rijnard van Tonder, Zack Coker, Deborah Katz, Mauricio Soto, Cody Kinneer, Jeremy Lacomis, Leo Chen, Trenton Tabor, and Milda Zizyte for sitting through many practice talks and providing invaluable feedback. And finally, thanks to all other researchers and friends at Carnegie Mellon for sharing their knowledge and expertise with me. This journey has been exciting and fulfilling thanks to all of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robotic systems are systems that sense, process, and physically react to information from the real world [138]. In the past decade, robotic systems have become increasingly important in our everyday lives. In the past, the use of these systems were mostly limited to industrial settings, in isolation and under specific safe conditions, which prevented potential extreme damages to people. However, robotic systems are now frequently used in a variety of (unsafe) settings such as avionics, transportation and medical operations. In response to the COVID-19 crisis, for example, robots have been used to deliver food to quarantined patients, disinfect public places, and cope with increased supply chain demands [42, 46, 256, 263]. It is now more important than ever to ensure the safety and quality of these systems before deploying them as failures in these systems can be catastrophic. In October 2018 and March 2019, two separate Boeing 737 MAX airplanes crashed after an uncommanded aggressive dive caused by erroneous angle of attack sensor data, killing all 346 people aboard [2].

Automated quality assurance, or more specifically automated testing, is widely used in software development [47]. However, robotic systems have specific properties that make deployment of automated testing challenging in practice: 1) robots are comprised of hardware, software, and physical components, which can be unreliable and non-deterministic [89, 133, 176], 2) they interact with the physical world via inherently noisy sensors and actuators, and are sensitive to timing differences [176], 3) they operate within the practically boundless state space of reality, making emergent behaviors (i.e., corner cases) difficult to predict [89], and 4) the notion of correctness for these systems is often non-exact and difficult to specify [192]. As a result, in many cases the integration and system-as-a-whole testing is performed manually, mostly in the field [20]. Field testing is an important part of robotics development, but it can result in expensive and dangerous failures. In addition, field testing is highly limited by the scale of the scenarios and environments to which it can be applied. For example, testing an autonomous drone under highly windy conditions requires either an expensive setup to artificially recreate high speed wind, or for the condition to happen naturally. None of these options are practical, which leaves features untested or under-tested in practice.

The ultimate goal of this work is to make robotics and cyberphysical systems safe, and improve the quality assurance of these systems. For this purpose, we first need to identify the most common challenges faced by robotics developers when testing robotic systems. Even though many studies have investigated the state of testing (especially automated testing) of software

Figure 1.1: Automated testing pipeline for a cyberphysical system using simulation. An automated test input generation technique (Chapter 5) creates test inputs that will be executed on the CPS in simulation, which results in execution traces. These traces are provided to an automated oracle (Chapter 4) to be labeled as either correct or erroneous.

systems in practice [59, 86, 106, 212, 240], little attention has been paid towards automated testing of robotics and cyberphysical systems. As part of this thesis, I conducted a qualitative study with robotics practitioners, to better understand the robotics testing practices currently being used, and identify challenges and bottlenecks preventing roboticists from automated testing [20]. I identified three themes of challenges in testing robotic systems: 1) real-world complexities, 2) community and standards, and 3) component integration. I present this study in greater detail in Section 3.1.

Even though hardware testing is an essential part of quality assurance for robotic systems [15, 261], it is (almost) orthogonal to the quality of operating software; if the behavior of the hardware and the environment can be simulated perfectly, we no longer need the actual hardware and a real environment to test the software of the system. However, simulation by definition provides an abstraction of the real environment, and most simulators provide low fidelity simulation and are very limited [71, 73, 293].

We find that robotics practitioners generally distrust low-fidelity simulation, and believe the fidelity of simulation is not sufficiently high for testing and that running tests on the actual robot is the only way to test the system [20, 291]. However, this common belief is not necessarily correct or supported by evidence. In 2016, the ExoMars lander crashed on Mars, which cost approximately $350 million [13]. Interestingly, this extremely expensive failure was later recreated in simulation, which shows the potential role that simulation-based testing could play in preventing failures in the field [267]. In a similar case, a report issued by the National Transportation Safety Board (NTSB) on Boeing 737 MAX crashes illustrates that the specific failure modes that could lead to uncommanded plane dive (e.g., erroneous sensor data) were not properly simulated as part of functional hazard assessment validation tests, and as a result, were missed by NTSB's safety assessment [3]. Although low-fidelity software-in-the-loop (SITL) simulators are not perfect, they can still be very effective tools in detecting and preventing failures by allowing cheap, large-scale, automated testing. To illustrate the extent to which low-fidelity simulation-based testing may be used to detect failures in robotics systems, we conducted an empirical case study on a robotic system [265]. In this study, presented in Section 3.2, we showed that more than half of the software bugs found in the system over time can manifest in low-fidelity SITL simulation. Specifically, we found that of all bugs, only 10% require particular environmental conditions (not available in low-fidelity simulation) to manifest, and 4% only manifest on physical hardware.

As a result, in the absence of high-fidelity simulators, low-fidelity SITL simulation can be used for systematic, large-scale automated testing of CPSs, as its low fidelity only prevents discovering of a small number of bugs in the system. However, prior work on the challenges of

testing in robotics [20, 183] and CPSs in general [292] broadly identifies simulation as a key element of testing that requires improvement. As part of this thesis, I set out to understand the extent to which simulation is used for testing and automated testing in practice, and identify the barriers that prevent wider use. For this purpose, we conducted a survey of 82 robotics developers to understand how they perceive simulation-based testing and what challenges they face when using simulators. We found that *simulation is used extensively for manual testing,* especially during early stages of design and development, but that *simulation is rarely used for automated testing.* Overall, we identified 10 challenges that make it difficult for developers to use simulation. I present this study in Section 3.3.

Given the considerable potential of simulation-based testing, we can imagine a fully automated testing pipeline that automatically performs large scale, whole-system tests on robotic and CPS software in simulation. Figure 1.1 presents a high-level depiction of such a pipeline. Overall, identifying failures and unexpected behaviors in the system requires 1) *triggering* the conditions and scenarios that result in the failure manifesting, and 2) a means of *detecting* the erroneous or unexpected system behaviors. In this thesis, I present techniques to achieve this pipeline by creating an automated oracle that focuses on *detecting* misbehaviors of the system (Chapter 4), and an automated test input generation technique that targets *triggering* faults in the system (Chapter 5), without requiring pre-defined specifications or models of the system.

SITL simulation-based testing, as any other testing method, requires an oracle that can differentiate between correct and incorrect system behavior [41]. This oracle can take many forms, from formal specifications to manual inspection (human judgment) [19, 149, 182, 188, 225, 296]. For large-scale, automated testing, we require an oracle that can *automatically* label executions of the system and detect failures. However, manually defining such oracle for a robotics system requires extensive knowledge about the (usually very complex) system and considering all possible scenarios that the robot may face in advance [199]. For example, the oracle for a self-driving car may simply specify that the vehicle should not collide with any objects. Even though this oracle can detect failures in conditions where the vehicle hits a pedestrian or an object, it does not take into account cases where colliding with an object, such as a plastic bag in the air, does not impose any danger and should be allowed. In other words, robotic systems' correct behavior may vary based on the conditions that are affected by the unpredictable environment, system's configurations, timing and randomness. An accurate oracle needs to consider all possible conditions and specify the correct behavior of the system in those conditions [162].

An automated approach of generating the oracle can take us one step closer to a pipeline for systematic, large-scale automated testing presented in Figure 1.1. Existing approaches in automated specification mining and invariant inference, formal verification and statistical models have tackled this problem in the past [29, 48, 50, 74, 77, 78, 88, 91, 112, 135, 172, 173, 213, 214, 216, 285, 296]. In Chapter 2, I present these techniques in greater detail, and discuss their advantages and limitations. Broadly, robotic and autonomous systems have features that limit the application of existing approaches. One such feature is that many of these systems involve third-party components without access to the source code. This heavily limits the application of many existing tools as they require complete access to the source code to extract system's specifications. Another feature of robotics systems, as already mentioned, involves the context-dependent behavior of the system, which eventually requires a method that is able to learn disjunctive models [213], depending on context. Finally, the amount of noise and randomness in the environment

in which these systems operate requires special attention.

My key insight is that by observing many executions of the system as a blackbox, we can find patterns of correct or normal behavior of the system, and mark executions that do not comply with these patterns as abnormal or erroneous behavior. I present my approach in generating these blackbox models from a set of previously observed executions of the system, and use them as an oracle that can differentiate between correct and erroneous behavior of the system. I evaluated the accuracy of these models by their ability to correctly label a set of traces. The details to this work is presented in Chapter 4.

In addition to being able to *detect* failures in the system, we need to effectively *trigger* the faulty behavior. Studies have shown that the quality of test inputs have a high impact on the ability to expose the system's faulty behaviors [141, 146, 222]. In Chapter 5, I study the navigation planner of a robotic system to develop an understanding of how we can evaluate the quality of test inputs, and to propose an approach to automatically generate effective test cases.

To get one step closer to the automated testing pipeline of Figure 1.1, I propose an automated method of generating effective test inputs that increases the ability of the testing framework to reveal faults in the system [156]. Prior studies have proposed a number of automated test input generation approaches to address this problem [110, 122, 197, 205, 268, 269, 277]. However, many of these approaches require pre-defined artifacts and models of the system (e.g., Simulink and MATLAB models) [122, 197, 205, 277], which are difficult and error-prone to be specified [110], and may not be available for many non-safety-critical robotic and CPSs [122]. Other approaches specifically focus on autonomous driving applications with a set of assumptions and requirements (e.g., the definition of safe driving that includes traffic laws) that may not be easily extendable to other systems [110, 268, 269]. In this thesis, I study and evaluate different characteristics of the test inputs that impact their quality, and propose an automated, search-based test generation approach to generate high-quality test inputs without requiring pre-defined models or artifacts of the system.

To generate effective test inputs, we require the following pieces: 1) A way to automatically specify and generate test input scenarios for simulation, which includes the specification of the simulation environment, and the test mission to be performed, 2) a metric that can measure the effectiveness of test inputs in revealing faults, and 3) an approach to automatically generate test inputs with higher effectiveness.

I addressed the first requirement by developing a tool to automatically translate test scenarios provided in a domain-specific language [97] to valid simulation scenes and missions. Second, to develop an understanding of how we can evaluate the effectiveness of test inputs, I did the following: 1) investigated the reliability of test outcomes with respect to deterministic test executions for different test inputs, as it affects the quality of tests [53, 93, 127, 170, 184], and how we evaluate them, and 2) using the mutation score [76] as the ground-truth measure for the effectiveness of test inputs [141, 222], I evaluated different coverage-based test input quality metrics. I observed high levels of non-determinism among test executions, which indicates the necessity to execute test inputs multiple times and look at the outcome of the test executions collectively. To address the third requirement, I showed that, in general, most coverage metrics are poor indicators of fault finding effectiveness, which matches with findings of prior studies on conventional software [57, 96, 107, 125, 137]. However, I found that the *union coverage*, the collection of all lines or branches executed over multiple runs of the same test, has a higher

correlation with the mutation score, and is a better indicator of test effectiveness.

Informed by the previous findings, I propose an evolutionary-based test generation approach with a fitness function that is based on *scenario coverage*, where the test generation or selection approaches focus on maximizing the diversity and effectiveness of the scenarios presented to the system under test [102, 150, 202, 233, 281]. I showed that scenario coverage metrics inspired by very limited knowledge of the system, together with an evolutionary algorithm, can be effective in generating high-quality test inputs with low cost.

Overall, by identifying the challenges in testing robotic and cyberphysical systems, and proposing approaches that address a subset of those challenges, this dissertation takes us one step closer towards large-scale, automated testing of these systems, which eventually results in higher quality systems.

1.1 Thesis Statement

Robotic and cyberphysical systems have unique features such as interacting with the physical world and integrating hardware and software components, which creates challenges for automated, large-scale testing approaches. An automated testing framework using software-in-the-loop (low-fidelity) simulation can facilitate automated testing for these systems. This framework can be offered using a clustering approach as an automated oracle, and an evolutionary-based automated test input generation with scenario coverage fitness functions.

1.2 Contributions

This thesis contains a set of qualitative and quantitative studies, where I conducted interviews and surveys to empirically study the challenges of testing and automated testing robotic and cyberphysical systems, and used grounded theory [61] to analyze the data. In addition, I conducted a case study on bugs in popular open-source ARDUPILOT system.

In this thesis, I propose approaches in automatically generating oracles, and effective test inputs for these systems. I use multiple popular open-source robotic systems to evaluate performance of the proposed techniques both in terms of automatically creating more accurate oracles, and generating more fault revealing test inputs.

This thesis contributes in the following ways:

1. It identifies the challenges of automated testing for robotics systems and discovers the practices currently being used in the field of robotics.
2. It shows that simulation-based testing can be an effective approach in identifying faults in these systems.
3. It identifies the challenges of using simulators for the purpose of (automated) testing, and the most prominent issues with currently available simulators.
4. It presents a black-box approach to automatically infer oracles for these systems based on observed executions of the robot in the simulated environment.

5. It investigates the severity of non-determinism among test executions in simulation, and offers insight on the performance of coverage-based quality metrics as indicator of test inputs fault-revealing effectiveness.
6. It presents an evolutionary-based automated test generation approach using scenario coverage as fitness function.

Additionally, this thesis contributes by publicly providing the following set of tools and datasets:

1. A dataset of bugs in the ARDUPILOT system, Dockerfiles used to construct the images for each bug, and a full description of each bug's characteristics: `https://github.com/squaresLab/ArduBugs`.
2. The codebook, questionnaire, and other material of a large-scale survey with robotics practitioners on the challenges of using robotic simulators for testing: `https://doi.org/10.5281/zenodo.4444256`.
3. An implementation of Mithra, my proposed automated oracle learning approach, and an implementation of a state-of-the-art competing technique: `https://bit.ly/2S9m7cd`.
4. A dataset of execution traces collected over executing missions on two sample robotic systems in simulation, a set of bugs for each system, and execution traces reflecting those bugs: `https://bit.ly/2S9m7cd`.
5. An implementation of GzScenic,my tool to automatically translate test scenarios provided in Scenic [97] domain-specific language to valid simulation scenes and missions in the popular, general-purpose Gazebo simulator: `https://github.com/squaresLab/GzScenic`.

Overall, this thesis identifies the challenges in testing robotic and cyberphysical systems, and proposes approaches that address a subset of those challenges, such as automated oracle inference, and automated test input generation. Together, these approaches can takes us one step closer towards large-scale, automated testing of these systems, which eventually results in higher quality systems.

Parts of this thesis have been published in peer reviewed venues, and parts are currently under review:

- **Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?**, Christopher S. Timperley, Afsoon Afzal, Deborah Katz, Jam Marcos Hernandez, and Claire Le Goues, in International Conference on Software Testing, Validation and Verification (ICST), 2018 [265].
- **A Study on Challenges of Testing Robotic Systems**, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher S. Timperley, in International Conference on Software Testing, Validation and Verification (ICST), 2020 [20].
- **Simulation for Robotics Test Automation: Developer Perspectives**, Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher S. Timperley, in International Conference on Software Testing, Validation and Verification (ICST), 2021 [21].
- **Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems**, Afsoon Afzal, Claire Le Goues, Christopher S. Timperley, Under Journal Review.

- **GzScenic: Automatic Scene Generation for Gazebo Simulator**, Afsoon Afzal, Claire Le Goues, Christopher S. Timperley, Under Review [22].

1.3 Outline

I first provide an overview of literature on the topics related to this thesis, and background on the related topics (Chapter 2). Chapter 3 presents the empirical studies on challenges of testing robotic and cyberphysical systems. Chapter 4 presents an automated oracle inference approach using clustering. In Chapter 5, I discuss automated test input generation, and different characteristics of robotic systems that complicate it. Finally in Chapter 6, I conclude this thesis.

Chapter 2

Review of Literature and Background

The following sections give an overview of related work and background that inform this thesis.

2.1 Related work

Robotic systems Robots are systems that sense, process, and physically react to information from the real world [138]. Robotic systems are a subcategory of cyberphysical systems [155], which include non-robotics systems such as networking systems or power grids. However, robotic systems are subject to system constraints that do not apply to CPSs broadly (such as a need for autonomy, route planning, and mobility).

Robotic systems differ in several important dimensions [83, 89, 133, 176, 192, 248] as compared to conventional software: (1) Robots are comprised of hardware, software, and physical components, which can be unreliable and non-determinstic [89, 133, 176]. (2) Robots interact with the physical world via inherently noisy sensors and actuators, and are sensitive to timing differences [176]. (3) Robots operate within the practically boundless state space of reality, making emergent behaviors (i.e., corner cases) difficult to predict [89]. (4) For robotic systems, the notion of correctness is often non-exact and difficult to specify [192]. These characteristics introduce unique challenges for testing, such as the need to either abstract aspects of physical reality or conduct extensive testing in the real world.

Challenges of testing robotics and CPSs A number of studies have investigated software testing practices broadly, and the challenges facing these practices [59, 86, 106, 212, 240]. Runeson [240] conducted a large-scale survey on unit testing with 19 software companies, and identified unit test definitions, strengths, and problems. Causevic et al. [59] qualitatively and quantitatively study practices and preferences on contemporary aspects of software testing.

In a technical report, Zheng et al. [291] report on a study of verification and validation in cyberphysical systems. The paper finds that there are significant research gaps in addressing verification and validation of CPS, and that these gaps potentially stand in the way of the construction of robust, reliable and resilient mission-critical CPS. The paper also finds that developers have a lack of trust in simulators, and one of the main research challenges they identify is integrated simulation. Seshia et al. [248] introduce a combination of characteristics that define

the challenges unique to the design automation of CPSs. Marijan et al. [192] speculate over a range of challenges involving testing of machine learning based systems. Garcia et al. [104] conduct a large-scale empirical study to assess the state of the art and practice of robotics software engineering in the service robotics domain.

Duan et al. [83] extract 27 challenges for verification of CPSs by performing a large-scale search on papers published from 2006 to 2018. Alami et al. [26] study the quality assurance practices of the Robot Operating System (ROS)[1] community by using qualitative methods such as interviews with ten participants, virtual ethnography, and community reach-outs. They learn that implementation and execution of QA practices in the ROS community are influenced by social and cultural factors and are constrained by sustainability and complexity. However, their results only apply to a specific robotics framework and cannot be generalized to non-ROS systems.

Luckcuck et al. [183] systematically surveyed the state of the art in formal specification and verification for autonomous robotics, and identified the challenges of formally specifying and verifying (autonomous) robotic systems. Their study focuses on formal specification as a method of quality assurance and does not provide information regarding other testing practices within the wider field of robotics.

Wienke et al. [275] conducted a large-scale survey to find out which types of failures currently exist in robotics and intelligent systems, what their origins are, and how these systems are monitored and debugged. Sotiropoulos et al. [254] performed a study of 33 bugs in academic code for outdoor robot navigation. The study found that for many navigation bugs, only a low-fidelity simulation of the environment is necessary to reproduce the bug. Garcia et al. [103] study 499 bugs in autonomous vehicles and classify those bugs into 13 root causes, 20 bug symptoms, and 18 categories of software components those bugs often affect. Koopman and Wagner [162] highlight the challenges of creating an end-to-end process that integrates the safety concerns of a myriad of technical specialties into a unified approach. Beschastnikh et al. [49] looked at several key features and debugging challenges that differentiate distributed systems from other kinds of software.

Anomaly detection There are a number of studies on anomaly detection in cyberphysical systems [66, 108, 120, 130, 210, 224, 272, 282, 296]. He et al. [124] proposes an approach for creating autoregressive system identification (AR-SI) oracles for CPSs. Based on the assumption that many CPSs are designed to run *smoothly* when noises are under control, AR-SI automatically determines whether a trace is erroneous or correct by checking the smoothness of the system's behavior. Theisslet et al. [262] propose an approach that reports anomalies in the multivariate time series, which point the expert to potential faults. Stocco et al. [258] propose an anomaly detection approach for Deep Neural Networks (DNNs) of autonomous driving systems that uses autoencoder- and time series-based anomaly detection to reconstruct the driving scenarios seen by the car, and to determine the confidence boundary between normal and unsupported conditions.

Chen et al. [65] build models by combining mutation testing and machine learning: they generate faulty versions (mutants) of the tested system and then learn SVM-based models using supervised learning over the resultant data traces corresponding to system execution. They

[1] `https://ros.org`

evaluate on a model of a physical water sanitation plant. Ghafouri et al. [109] show that common supervised approaches in this context are vulnerable to stealthy attacks. An unsupervised technique [136] evaluated on the same treatment plant model trains a Deep Neural Net (DNN) to identify outliers. Ye et al. [288] use a multivariate quality control technique to detect intrusions by building a long-term profile of normal activities in information systems and using the norm profile to detect anomalies.

Other approaches target the detection of particular attack classes specifically. Choi et al. [67] present a technique that infers control invariants to identify external physical attacks against robotic vehicles. Alippi et al. [30] learn Hidden Markov Models of highly correlated sensor data that are then used to find sensor faults. Abbaspour et al. [14] train adaptive neural networks over faults injected into sensor data to detect fault data injection attacks in an unmanned aerial vehicle.

The oracle problem Fully automated testing for CPSs requires oracles that can determine whether a given CPS behaves correctly for a given set of inputs [41]. In typical research and practice, domain experts manually provide CPS oracles in the form of a set of partial specifications, or assertions [19, 149, 182, 188, 225, 296]. However, manually writing such specifications is tedious, complex, and error-prone [110, 199].

A number of techniques proposed approaches for inferring invariants or finite state models describing correct software behavior perform what is known as dynamic specification mining. Existing dynamic specification mining techniques can be classified into four categories based on the kind of models that they produce: data properties (a.k.a. invariants) [74, 88, 112, 213, 214], temporal event properties [48, 50, 173, 285], timing properties [216, 246], and hybrid models [29, 172, 216] that combine multiple types of model. These techniques are generally poorly-suited to the CPS context. Most require source code access or instrumentation, and none are suitable for time series data. Techniques like Daikon [88] and its numerous successors (e.g., DySy [74], SymInfer [214], Nguyen et al. [213], or Dinv [112], among others) learn source- or method-level data invariants rather than models of correct execution behavior. Techniques like Texada [173] and Perracotta [285] do learn temporal properties between events but do not model or learn temporal data properties, a key primitive in CPS execution (Artinali [29] comes closest to this goal, learning event ordering and data properties *within* an event).

As another way of approaching the oracle problem for CPSs, studies have used metamorphic testing to observe the relations between the inputs and outputs of multiple executions of a CPS [179, 264, 294]. Lindvall at al. [179] exploit tests with same expected output according to a given model to test autonomous systems. Zhou and Sun [294] use metamorphic testing to specifically detect software errors from the LiDAR sensor of autonomous vehicles. Tian et al. [264] introduce DeepTest, a testing tool for automatically detecting erroneous behaviors of DNN-driven vehicles. As an oracle, they use metamorphic testing by checking that properties like steering angle of an autonomous vehicle remain unchanged in different conditions such as different weather or lighting. Menghi et al. [201] propose an automated approach to translate CPS requirements specified in a logic-based language into test oracles specified in Simulink's simulation language for CPSs.

Automated Test generation Software test automation significantly improves the quality, and automated test suite generation significantly affects the software test automation, and is a very integral part of the automation process [156]. Automated test suite generation for CPSs includes numerous model-based approaches [16, 58, 197, 205, 277]. These approaches require a model of the system in a particular format (e.g., Simulink or MATLAB models) to generate a set of test cases that reach the highest coverage of the model. In the absence of such models, search-based techniques have shown promise in automated test suite generation of CPSs [110, 122, 268, 269].

Search-Based Software Testing (SBST) is a method for automated test generation based on optimization using meta-heuristics [27, 198]. The SBST approaches require a fitness function that has crucial impact on their performance [27, 31, 243]. In a study on Java programs, Salahirad et al. [243] showed that fitness functions that thoroughly explore system structure should be used as primary generation objectives, supported by secondary fitness functions that explore orthogonal, supporting scenarios. Fraser et al. [95] propose a novel paradigm in which whole test suites are evolved with the aim of covering all coverage goals at the same time while keeping the total size as small as possible.

Arrieta et al. [35] propose a search-based approach that aims to cost-effectively optimize the test process of CPS product lines by prioritizing the test cases that are executed in specific products at different test levels. By applying SBST to automated driving controls, Gladisch et al. [110] show that SBST is capable of finding relevant errors and provide valuable feedback to the developers, but requires tool support for writing specifications. Bagschik et al. [38] propose a generation of traffic scenes in natural language as a basis for a scenario creation for automated vehicles. Similarly, Gambi et al. [100] recreate real car crashes as physically accurate simulations in an environment that can be used for testing self-driving car software. Haq et al. [119] show that simulator-generated datasets or test inputs for testing DNNs in automated driving systems have similar performance as to those obtained by testing DNNs with real-life datasets.

To take uncertainty that is unavoidable in the behaviors of CPSs into consideration at various testing phases, including test generation, Ali et al. [28] propose uncertainty-wise testing, arguing that uncertainty (i.e., lack of knowledge) in the behavior of a CPS, its operating environment, and in their interactions must be explicitly considered during the testing phase. Hutchison et al. [133] outline a framework for automated robustness testing of autonomy systems that builds on traditional robustness testing, drawing from a dictionary of exceptional values to construct test inputs to systems and components.

Test Input Quality Metrics Mutation testing is a fault-based testing technique which provides a testing criterion called the *mutation score* [76]. The mutation score is calculated by executing a test suite on a set of mutated programs (i.e., programs that are injected with faults), and measuring the number of these executions that result in different outputs than running the same test suite on the original, unmutated program. The program can be mutated using different mutation operators that differ in their impact on the program [139, 215]. The mutation score can be used to measure the effectiveness of a test set in terms of its ability to detect faults [141]. Achieving higher mutation scores improves the fault detection significantly [222]. On safety-critical systems mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed [39].

Besides mutation score, studies have used different coverage-based metrics (e.g., function coverage, statement coverage, branch coverage) as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [131, 185, 230, 259]. Automated test generation tools and approaches for conventional software commonly use coverage metrics as a mean of evaluating their test suites, and attempt to generate test suites that maximize the coverage metrics [95, 177, 223, 287]. Prior studies have investigated the effectiveness of coverage metrics, and have shown that these metrics are generally poor indicators of fault finding effectiveness [57, 96, 107, 125, 137].

In addition to code coverage metrics, neuron coverage [121] and model coverage [284] metrics have been specifically proposed for deep neural networks that are commonly used in robotic and CPSs software. In testing robotic and CPSs, we desire to expose the SUT to different, possibly all scenarios and situations that the system can be faced to ensure that the system performs as expected, in a safe manner. In the field of autonomous vehicles (AV), this metric is known as *scenario coverage* or *situation coverage* where the test suites have higher quality if they cover a diverse set of scenarios and situations [34, 102, 123, 150, 202, 208, 233, 270, 281]. Xia et al. [281] create an influence factor and importance degree model for different elements of a driving scenario such as the environment (e.g., weather), positioning of the roads, and the road traffic, which they use to generate testing scenarios that are more effective in challenging the driving control system, and are diverse. Similarly, Arcaini et al. [34] introduce the notion of patterns of driving characteristics in an autonomous driving system, to characterize their interaction and measure their duration.

Flaky Tests and Non-determinism Many software systems (e.g., distributed systems, CPSs, embedded systems) exhibit a level of non-determinism in their behavior, meaning that running the exact same inputs under the exact same conditions may result in different outputs and behaviors [53, 93, 127, 170, 184]. Overall, non-determinism in the system has serious ramifications for testing including *flaky tests* where a single execution of the test inputs is not sufficient to mark the test as passing or failing [43]. Prior studies have investigated flaky tests in conventional software systems extensively, and have introduced approaches to automatically identify flaky tests in a test suite [43, 94, 115, 118, 165, 166, 167].

2.2 Background

ArduPilot The open-source ARDUPILOT project[2], written in C++, uses a common framework and collection of libraries to implement a set of general-purpose autopilot systems for use with a variety of vehicles, including, but not limited to, submarines, helicopters, multirotors, and airplanes (Figure 2.1). ARDUPILOT is extremely popular with hobbyists and professionals alike. It is installed in over one million vehicles worldwide and used by organizations including NASA, Intel, and Boeing, as well as many higher-education institutes around the world.[3]

I use ARDUPILOT as one of my case studies due to being highly popular and open-source, and its rich version-control history, containing over 30,000 commits since May 2010, and for its

[2] `http://ardupilot.org`
[3] `http://ardupilot.org/about`

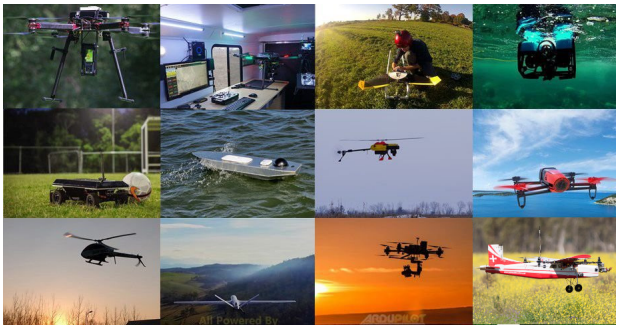


Figure 2.1: A variety of cyberphysical systems, including airplanes, helicopters, and submarines, that use ArduPilot as an autopilot software. (Source: `https://ardupilot.org`)

consistent bug-fix commit description conventions. ARDUPILOT has been widely used in studies on CPSs as it represents a fairly complex open-source CPS [18, 124, 179, 274, 296], and contains 300,000 lines of code (measured using SLOC).

To facilitate rapid prototyping and reduce the costs of whole-system testing, ARDUPILOT offers a number of simulators for most of its vehicles (excluding submarines). In general, those platforms simulate the dynamics of the vehicle under test, feed artificial sensor values to the controller, and relay the state of its actuators to the physics simulation. Hardware-in-the-loop (HIL) simulators are used to perform testing on a given flight controller hardware device by directly reading from and writing to it. In contrast, software-in-the-loop (SITL) simulators test a software implementation of the flight controller by running it on a general-purpose computer.

Robot Operating System (ROS) The Robot Operating System (ROS) [229] is a flexible framework for writing robot software provided by Open Robotics. [4] It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS follows a publisher-subscriber architecture, where nodes are processes that perform computation, and they communicate with each other by passing messages. A node sends a message by publishing it to a given topic, which is simply a string such as "odometry" or "map". A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence.

ROS is a relatively young framework (first released in 2009), and is currently used by thousands of people around the world; ROS has more than 34,000 registered users on *ROSAnswers*, the main Q&A platform for ROS users.[5] There are two main versions of ROS (i.e., ROS 1 and ROS 2), and they follow an annual release model that is both similar to and linked to Ubuntu, and to this day, there have been 13 official, released ROS 1 distributions (e.g., Neotic, Melodic, Lunar, and Kinetic), and 6 ROS 2 distributions.

ROS is designed with the purpose of encouraging *collaborative* robotics software development by allowing robotics developers to build upon each other's work [218]. In fact, a number

[4] `https://ros.org`
[5] `http://download.ros.org/downloads/metrics/metrics-report-2019-07.pdf`

of ROS packages provided by Open Robotics and other working groups that offer fundamental building blocks required to construct most robots (e.g., navigation, perception, drivers) are highly popular and are almost unanimously used by all ROS-based systems [160].

In this thesis, I use two ROS-based robotic systems to demonstrate the effectiveness my approaches on these systems. I use the F1/10 system [217], which is an open-source, autonomous racing cyberphysical platform, one tenth of the size of a real Formula 1 racing car, that is built on top of ROS, and designed to be used as a testbed for research and education. Additionally, I use TURTLEBOT3 [10], a programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping [25, 32, 227]. TURTLEBOT3 can be customized into various ways depending on how you reconstruct the mechanical parts and use optional parts such as the computer and sensor.

Robotic Simulators Robotic simulators model the physical aspects of the robot (e.g., kinematics), its operating environment (e.g., terrain, obstacles, lighting, weather), and the interaction between them. The simulator provides synthetic sensor readings to the robot controller at a fixed interval and listens to actuation signals from the robot controller (e.g., electronic speed control signals). The simulator performs a stepwise simulation of the virtual world that models certain physical interactions and phenomena at discrete time steps.

The two most popular forms of simulation-based testing are hardware-in-the-loop testing (HIL) and software-in-the-loop testing (SITL). During HIL, the robot controller software runs on the robot's embedded hardware that is connected to a simulator that typically runs on a separate machine with greater resources (e.g., dedicated GPU). SITL, on the other hand, runs the robot controller software on the same machine as the simulator. HIL is typically slower and more expensive than SITL but can be used to test the integration between the robot controller's software and embedded hardware. In this thesis, I exclusively focus on SITL simulation.

Numerous robotics simulators are available, each with different characteristics. General purpose simulators, such as Gazebo [159], CoppeliaSim (formerly known as V-REP) [237], Unity [144], and MuJoCo [266], can be used to model a wide range of systems. Others, such as CARLA [82], LGSVL [175], AirSim [249] and AADS [178], are specifically designed for a certain robotics domain. These simulators differ vastly in their capabilities, features, and performance [75, 87, 228, 255]. Gazebo in particular is a popular, general-purpose robotics simulator[134, 159], maintained by Open Robotics, that has been used in a wide variety of domains and is the *de facto* simulation platform used by ROS.

Chapter 3

Challenges of Testing Robotic Systems

As described in Chapter 2, robotics and cyberphysical systems have features such as non-deterministic behavior and noisy sensors that make them different from conventional software systems. These features can specifically create challenges for automated testing approaches. Many studies have focused on understanding the challenges of testing in conventional software systems, but limited attention has been paid to robotics.

To better understand the state of (automated) testing in robotics, my collaborators and I conducted a series of qualitative and quantitative studies. We first identified challenges of (automated) testing in the field of robotics, and testing practices currently being used in this field by conducting a series of qualitative studies with robotics practitioners. In this study, we identified 9 challenges that robotics practitioners face while testing their robotic systems [20].

Secondly, we investigated the potential impact of using low-fidelity software-based simulation on exposing failures in robotics systems by conducting a case study. In this study, we showed that low-fidelity simulation can be an effective approach in detecting bugs and errors with low cost in robotic systems [265].

Finally, as features of SITL simulators highly impact the automated testing of these systems, we conducted a large-scale survey with robotics practitioners to identify features in robotics simulators that are the most important for automated testing, and the challenges of using these simulators. In this study, we identified 10 challenges that make it difficult for developers to use simulation in general (i.e., for any purpose), for testing, and specifically for automated testing [21].

3.1 Testing in Robotics: Practices and Challenges

As described in Chapter 2, unique features of robotics and cyberphysical systems such as interaction with the real world through noisy sensors and actuators, introduce challenges to automated testing and validation of these systems. However, no prior studies have identified testing practices and challenges for robotic systems. Our goal in this study is to gain an in-depth understanding of existing testing practices and challenges within the robotics industry.

We conduct a series of qualitative interviews with 12 robotics practitioners from 11 robotics companies and institutions. Specifically, we investigate the testing practices that are being used

in the field of robotics, and the challenges faced by roboticists when testing their systems. We answer the following research questions:

- **RQ1:** *What testing practices are currently being used by roboticists?*
- **RQ2:** *What are the costs and barriers to designing and writing tests for robotic systems?*
- **RQ3:** *What are the costs and barriers to running and automating tests in robotic systems?*

Having a better understanding of the current state of testing in robotics, as well as the problems and concerns of the robotics community regarding testing of robotic systems, will guide researchers and practitioners to provide and apply solutions that can ultimately result in higher-quality robotic systems. The result of this study is published in the International Conference on Software Testing, Verification and Validation (ICST 2020) [20].

3.1.1 Methodology

Interviews are useful instruments for getting the story behind a participant's experiences, acquiring in-depth information on a topic, and soliciting unexpected types of information [200, 251]. We developed our interview script by performing a series of iterative pilots.

We recruited our participants through a variety of means. Our goal was to select participants from a broad range of positions and to sample across a diversity of industries, company size, and experience. We recruited our first three participants using convenience sampling. We recruited the rest of our participants using snowball sampling and targeted messages to developers that we found on LinkedIn and Twitter who had the phrase "robotics engineer" in their profile.

Overall, we interviewed 12 robotics practitioners with a variety of backgrounds and experiences. These practitioners represent 11 robotics companies and institutions ranging from small startups to large multi-national companies. A summary of the relevant details of the participants of our study is presented in Table 3.1. After performing the interviews, we determined that while P5 and P7 work at a company that is heavily involved in robotics, both of the participants are focused on non-robotics-related software development, and so we removed them from our sample moving forward.

Interviews and coding We conducted semi-structured interviews that lasted between 30 to 60 minutes over phone, video chat, or in person. We prepared an interview script with detailed questions providing insight into our research questions. A subset of questions on the interview script are presented in Table 3.2. However, we only used the script to guide the interviews. We adjusted interview questions based on the experience of the participant to gain a deeper understanding of their testing practices and challenges. We took notes from interviewee responses and recorded the interviews with their consent to validate our notes. We then used a grounded, iterative approach to code our notes. We first labeled responses based on their relevance to our research questions. Then, we iteratively coded the notes based on common themes, discussed the codes and redefined them.

Validation To validate the results of our study and conclusions, we sent a full draft of the results to our participants. We asked participants to inform us of their level of agreement with

Table 3.1: Interview participants, their experience with robotics, their role in the company or institution, type and sector of their company or institution, and whether their testing process includes a dedicated quality assurance team.

	PARTICIPANT			COMPANY/INSTITUTE		
ID	Background	Experience (years)	Role	Type	Sector	QA team?
P1	Software Engineering	6	Developer	Startup	Mobile Services	✘
P2	Electrical Engineering	> 10	Principal Engineer	Academia	Research & Development	✘
P3	Embedded Software Engineering	2	Developer	Multinational Company	Autonomous Vehicle	✔
P4	Mechanical & Robotics Engineering	5	Developer	Research Lab	Agriculture	✘
P5	Software Engineering	> 10	Test Engineer	Multinational Company	Industrial Automation	✔
P6	Math & Physics	> 10	Project Manager	Startup	Education	✘
P7	Experimental Physics	7	Test Engineer	Multinational Company	Industrial Automation	✔
P8	Mechanical Engineering & Math	5	Manager/Engineer	Startup	Cleaning	✘
P9	Computer Science	4	Engineer	Robotics contractor	Research & Development	✔
P10	Computer Science	> 10	Research Engineer	Academia	Industrial Automation	✘
P11	Computer Science & Math	< 1	Software Engineer	Multinational Company	Industrial Automation	✔
P12	Robotics Engineering	> 10	CTO	Startup	Mobile Services	✘

our conclusions and to provide their thoughts on our results. In total, six of the participants responded to our request. Four responded in total agreement with the results. The other two participants that responded provided specific feedback on our interpretation of their responses, and we incorporated their feedback into the final version of this paper.

3.1.2 Results

In this section, I discuss the results in response to our research questions. Overall, we identified 12 testing practices in use by roboticists, and 9 challenges for robotics testing.

Table 3.2: Sample questions on the interview script.

Practice	• What are all the different types of testing you do? • Can you describe your test running/writing process? • How much of your testing is done for certification? • Which types of tests find the most problems?
Testing Challenges	• What is difficult about writing tests? • Have these difficulties ever made you giving up on writing the tests at all? • Is there any part of writing tests that is not difficult? • What types of tests do you have the most difficulty running? • In your experience, is there anything that helps with making it easier to run tests? • For your tests that are not fully automated, why are they not? • What tools/frameworks/techniques do you use to simplify running tests? • Do you use simulation?
General	• What do you think is the most important bottleneck in the way of testing in robotics? • How do you think the difficulties of testing in robotics differ from your other experiences in other software development domains?

RQ1: Testing practices in robotics To determine the testing practices that are used in the robotics industry, we asked our participants to describe their own testing practices. In total, our participants reported 12 different testing practices, summarized in Table 3.3. Given the explorative nature of our study, we do not make any claim about the popularity of the reported practices; rather, we aim to identify the variety of testing methods that are used in robotics. Below, I discuss a selection of identified practices, bolded in Table 3.3, in more detail.

(T1) Field testing A full system test that happens in an environment that is similar or identical to the intended deployment environment can reveal many problems, as a robot is exposed to real-world scenarios and input. According to our participants, field testing is a common practice in robotics. Several of our participants mentioned that they conduct field testing frequently during both development and testing of a robot. For example, P9 and P12 both mentioned one or two-week long field testing events that take place after each development cycle. P2 stated that they conduct field testing once or twice a week.

(T2) Logging and playback Logs that are collected during the operation of a system contain important information about system execution for testing, debugging, and development of algorithms. Five of our participants reported that they collect detailed logs during the operation of their systems. Some use recorded logs for debugging and monitoring. For example, P9 provided an example where their robot logs whenever it resets, and they automatically process the logs to ensure that no unexpected, silent reset took place during the operation of the robot.

Logs can also be used to playback events and sensor data (a.k.a. record-and-replay) by feeding input collected from previous operations (either in the field or in simulation) to a robot. For example, ROSBAG is a widely-used command-line tool that records and replays messages for

Table 3.3: A summary of the testing practices that were reported by participants. Practices in bold are discussed in more detail in the text.

ID	Title	Description
T1	**Field testing**	Full-system testing in a real-world environment that shares similarities with the deployment environment.
T2	**Logging and playback**	The use of logged data, collected in the field, for the purposes of testing, debugging, and development (a.k.a. record and replay).
T3	**Simulation testing**	Tests that are executed in a simulated environment that can be used for both testing and development.
T4	**Plan-based testing**	The practice of planning an adequate sequence of field tests for validating that the system meets its requirements given a fixed testing budget (e.g., time, hardware, cost).
T5	Compliance testing	Testing for the purposes of determining whether a system complies with certain standards.
T6	Unit testing	Small, automated tests for validating individual code-level software components (i.e., functions).
T7	Performance testing	Subjecting a system to various workloads to ensure that it meets its functional (e.g., localization accuracy) and non-functional requirements (e.g., timeliness, memory).
T8	Hardware testing	Testing for the purposes of assessing the quality and integrity of hardware components prior to software integration (e.g., testing sensors and cameras).
T9	Robustness testing	Testing the system under extreme boundary conditions (e.g., a malfunctioning sensor) that are usually artificially injected to determine the safe operating limits of the system.
T10	Regression testing	Ensuring that changes to the system (e.g., the addition of a new feature) do not negatively affect existing functionality in an unintended way.
T11	Continuous integration	The practice of continually and automatically rebuilding the system and executing some portion of its tests (e.g., unit tests) as changes are made.
T12	Test maintenance	The practice of refactoring and maintaining tests to eliminate false positives, flakiness and redundancy, and to reflect changes to the requirements of the system.

robots built using ROS.[1] P10 and P12, for example, use record-and-replay to collect test inputs and debug their robots. P2 mentioned using record-and-replay to develop algorithms for their robots:

> *We often do not know why robots are making the choices that they make. By playing back the data in testing we can see why the robot made the choice it did, and then tweak the algorithm to see how it changes the robots behavior.*

(T3) Simulation testing Using simulated environments (rather than real-world, physical environments) can beneficially reduce the cost of testing and increase the opportunities for test automation [254, 265]. However, few participants reported that they used simulation as part of their testing process, even though all participants were aware of the theoretical benefits of simulation. For instance, P12 specifically said:

> *Our best way to test [algorithmic modules that are very dependent on input data] is through simulation, but we don't. We test in the real-world.*

Participants report that simulation is sometimes used as a tool during development, especially for high-level algorithms such as planning. P2 mentioned that they use simulation to create artificial scenarios while developing an algorithm. P9 said that their software team uses simulation to facilitate software development before the hardware platform is available.

Although simulation testing provides additional opportunities for test automation, our participants rarely used simulation for this purpose. Only P3 mentioned using simulation to some extent for automated testing:

> *We have some simulation cluster, so we could setup a script to run a simulation test automatically.*

(T4) Plan-based testing An outline and objectives for testing can be specified in advance in order to manage and guide testing. Test plans can be created based on different criteria such as formally specified system requirements, and recently added/modified features. P2, P8, and P9 all create a system requirements list, which is used to ensure all components of the system are covered by the tests. For example, P9 said:

> *We have a requirements list we edit with our sponsor. We measure quality of tests against requirements coverage, not code coverage.*

However, in P3's company, developers provide a test plan and failure criteria to test engineers for newly added or modified features.

RQ2: Challenges of designing and writing tests We asked our participants to describe the challenges they face when designing and writing tests for their products. We identified four common themes of challenges from their responses, summarized in Table 3.4.

(C1) Unpredictable corner cases Robots are typically expected to operate in many different environments and conditions. In most cases, the robotic state space is infinite, since it interacts with the real world; predicting the exact behavior of the physical environment is not

[1] `http://wiki.ros.org/rosbag`

Table 3.4: A summary of the challenges of designing, running, and automating tests for robotics that we identified based on participant responses.

ID	Title	Description
C1	**Unpredictable corner cases**	The challenge of attempting to anticipate and cover for all possible edge cases within a large (and possibly unbounded) state space when designing tests.
C2	**Engineering complexity**	A disproportionate level of engineering effort is required to build and maintain end-to-end test harnesses for robotic systems with respect to the benefit of those tests.
C3	**Culture of testing**	The challenge of operating within a culture that places little value on testing and provides developers with few incentives to write tests.
C4	**Coordination, collaboration, and documentation**	A lack of proper channels for coordination and collaboration among multiple teams (especially software and hardware teams), and a lack of documentation.
C5	**Cost and resources**	The cost of running and automating the tests in terms of human hours, resources and setup, and running time.
C6	**Environmental complexity**	The inherent difficulties of attempting to account for the complexities of the real world when simulating, testing, and reproducing full-system behavior.
C7	**Lack of oracle**	The challenge of specifying an oracle that can automatically distinguish between correct and incorrect behavior.
C8	**Software and hardware integration**	Difficulties that arise when different software and hardware components of the system are integrated and tested.
C9	**Distrust of simulation**	A lack of confidence in the accuracy and validity of results obtained by testing in simulation and synthetic environments, and a sole reliance on field testing.

viable [28, 232]. Attempting to account for all possible conditions and scenarios when designing tests is extremely difficult, if not impossible. For example, a plastic bag flying in front of a self-driving car's sensor is a case that may not immediately come to mind when designing tests. However, these unexpected corner cases are often the cause of failures [40].

Even though this challenge of a vast input space with unpredictable corner cases is not specific to robotic systems, it can be more manageable in non-robotics, software systems. In software systems (e.g., a web application), well-defined interfaces control and limit the range of inputs that can be received from external sources (e.g., users). P12 elaborated:

Software systems need to communicate only within themselves and you can strongly define

the range of inputs that will come in. When a user is involved, the range of inputs grows, but it is limited by the range of inputs that can be produced by the user. When you have a physical system that needs to interact with the real world, you need to handle the vast state space and noise in the real world.

They later shared an example of this problem where the hardware for their robot was affected by very low temperatures in the field. At $-30°C$, some of the hardware started misbehaving and produced unexpected sensor data, which could lead to poor algorithmic decisions and unexpected behaviors. This event was something that had not been anticipated before it was actually witnessed in the field.

(C2) Engineering complexity The engineering effort required to prepare all pieces needed for testing a robot can be extremely high, as these systems can be very complex. All of our participants unanimously described their systems as extremely complex. P12 believes that robotics field is far away from deploying complex systems. They said:

As an industry, we haven't managed to deploy anything more complex than a Roomba, which basically operates using a one-dimensional input.

One aspect of engineering complexity involves the amount of scaffolding that is required to put the system into a testable state [133]. For example, P2 and P6 both consider it a challenge to write tests for incomplete components such as cases where the hardware of the system has not yet been fully designed or manufactured. P10 said:

Whenever working with network protocols, I see whether anyone has already written a protocol. If not, I'll start by creating a Wireshark plugin to debug the protocol before I start working on it.

Another engineering complexity affecting test design is the specification of test inputs. To design realistic inputs, roboticists sometimes need to collect data from the real world, which may be a challenge (e.g., a space rover). For instance, P4 mentioned the need to collect gigabytes of LIDAR data to reasonably test a small snippet of code.

Finally, the complexity of the system itself creates a challenge for roboticists to design and write tests that, as P9 puts it, "effectively validate all requirements for the system". P11 finds it difficult to understand what needs to be tested, and design tests that clearly signal failures and help the developers to identify the source of failures. They later mentioned that, based on their experience, writing tests can consume more time than the actual implementation. P12 believes that writing tests sometimes requires knowledge about many fields such as computer science, mathematics, and engineering.

(C3) Culture of testing Our participants referred to a culture of not believing in the value of testing in their company or institute among not only the roboticists, but also their sponsors and customers. For example, P4 mentioned that many developers do not see much practical value in unit tests, even though they theoretically understand the value of having them. The prevalence of such culture within a community may result in developers getting discouraged from writing effective tests. Both P4 and P9 mentioned being under pressure by their sponsors and clients to deliver the product as quickly as possible, and being discouraged from spending time on writing tests.

One of the characteristics of the robotics community is that it brings together people from many different disciplines (e.g., electrical and mechanical engineering). While the diversity of the community is a driving factor for many great advances in robotics, it can also introduce challenges. As P11 said:

The world of robotics unites folks from different back- grounds. Folks from a software background might observe testing differently from those who aren't.

Another cultural aspect of the robotics community that impacts testing practices pertains to the age of the industry and its associated startup culture that often values rapid prototyping and development over testing and quality assurance. P10 said:

The robotics community are more focused on making cool things than software quality and making things better.

The desire to be first to the market and having the robot with greatest number of features is often valued more than the quality and robustness of the robot.

Finally, we observed from the responses of our participants that there is often a high degree of reliance upon intuition during testing and development. P2, P4, P6, and P12 all specifically mentioned their intuition as an important tool for testing and debugging. For example, P2 said:

Personally, I have an intuition. I think I know what when something goes wrong.

(C4) Coordination, collaboration, and documentation In many robotics companies, significant coordination and collaboration is required to design meaningful tests for the system (especially for full system tests). This coordination can take place between separate development and testing teams, or between software and hardware teams. For example, both P2 and P3 found it challenging to integrate components developed by different teams and to coordinate final full-system testing after integration of software and hardware. A lack of documentation for third-party components adds further complexity to writing tests. Many robotic systems consist of third-party components for which full access to the source code is not granted. Furthermore, developers are often less familiar with such components since they did not develop those components themselves, and as such, they need to refer to documentation when designing tests involving third-party components. P10 faced this challenge when writing tests involving a third-party component without any form of documentation.

An additional challenge is that there are very few standards and guidelines for practitioners to guide their testing. P8 said:

A standard for robotic system testing would be neat. A process to follow. Like "here's how you assess a robotic system".

The more popular and advanced subfields of robotics (e.g., self-driving vehicles) are already beginning to provide standards and certification [84]. As P10 describes, "some areas of robotics have very crystalized safety regulations".

RQ3: Challenges of running and automating tests We asked our participants to describe the challenges they face when running tests on their systems, and the challenges of automating their tests. Below, I describe the five challenges, summarized in Table 3.4, that we extracted from participant responses.

(C5) Cost and resources There are several costs involved in running tests for robotics systems. First, conducting manual field testing can be dangerous and expensive. P10 described an accident where the robotic arm behaved unexpectedly, and crashed into the tester on their knee. They further said:

> *Once you experience a few accidents [during field testing], you realize that testing is really dangerous. If a typical software system like Excel crashes, no-one dies. For robotics, that certainly isn't the case.*

Second, developers and test teams need to spend many hours running test scenarios on the robot. The test team of P3's company receives tens to hundreds of test requests every day, but their time and resources (e.g., physical robots) are limited. Given limited time and resources, developers and testers are forced to select, prioritize, and minimize tests in order to test as many changes to the software or requirements as is possible. Third, it may take a long time to run the tests. P1, P6, P8, and P9 all find the long running time of tests as one of the challenges of testing.

Finally, the cost of the equipment and setup required for running tests may be prohibitively expensive for smaller companies. P10 said that their robots are so expensive that they need to be extremely careful when interacting with them. To be able to design large-scale automated tests for their robot, P8 needs to build a framework that can automatically capture the state of the robot. P8 believed that it is less expensive for their company to pay an intern to manually test the robot, rather than designing a computer vision platform that will allow them to write automated tests. P1 uses simulation for running tests, but finds simulation a bottleneck of their testing practices because of its low speed and the amount of resources required for running it.

Similarly, automating tests requires significant efforts and investment that may be considered too expensive in terms of developer hours and resources. P1 describes automating tests of specific hardware and network interactions as "too much work" as they need to use mocks and patches to imitate other components. P2 and P11 both claim that establishing an infrastructure for automated testing (via simulation) is very difficult and expensive. P8 and P9 believe that it is always possible to hit deeper levels of test automation as long as the cost is justifiable. For example, P8 said:

> *There's a trade-off between cost of automating tests and number of times that we have to run them. We don't need to run some tests very often, so we don't really need to automate them.*

(C6) Environmental complexity The intended operating environment for a robot can be very complex: robots are embodied within the unpredictable and practically boundless state of space of reality, and their behavior may be dependent upon certain physical features (e.g., terrain) and phenomena (e.g., lighting, weather). Finding a suitable environment for testing the robot under expected operating conditions (e.g., on Mars or in the deep ocean) can be challenging: P8 mentioned that a challenge they face is finding as many qualitatively different physical locations as they can to test their robot, since every environment may have characteristics that reveals problems in their system. However, these environments sometimes constrain the number and quality of tests that can be run.

The complexity of attempting to model physical reality also complicates the development of high-fidelity simulators, which are extremely important for both running and automating

tests [248, 291]. P12 believes that "no simulators currently exist where the information is even close to the reality, they are nowhere close to the noise and variability of real-world data". P2 and P10 also believe that simulation cannot provide sufficient fidelity for testing robots in realistic scenarios.

Finally, complexities of the real world can hinder the reproduction of bugs and certain tests. P6, P11, and P12 have all faced the challenge of reproducing bugs they discovered in the field. P12 used the term *Heisenbugs* [113] to describe these bugs that will only manifest when you are not looking for them. P11 believes that, even though record and replay has many benefits, it is not the ultimate solution to reproducibility since you need to make sure that the replayed state is true to the world (e.g., with respect to timestamps and orderings).

Record and replay was reported as a popular approach for dealing with the challenges of testing systems with complex environments (discussed earlier). Sensor data is recorded in the field and then replayed for testing purposes. This approach has advantages, in that it uses real data and it is often easier to collect data than to synthetically create large volumes of data. However, there are also significant limitations to this approach. Without enough varied data, developers can run the risk of overfitting their approach to the recorded data, which might not represent the true variety of environments the robot will operate in. Additionally, because of the non-interactive nature of record and replay, it cannot be used for testing scenarios where there must be a feedback loop between the robot and the environment. P11 said:

> *Simulators are expensive, especially if you have to write your own. The more you are trying to test interactions with the physical world, the more value you will see in simulation. If there is less interaction, then record and replay is preferable.*

(C7) Lack of oracle The well-known oracle problem concerns how to distinguish whether a given system behavior is correct or incorrect [41]. Fully automated tests require an oracle that can automatically determine the correctness of system behavior. Because of the noisy and non-deterministic nature of robotic systems, it is difficult to discretely specify the exact behavior that is intended [124]. For example, consider that a robot is instructed to move to a given position, but that the robot stops 5 centimeters away from the exact coordinates of its destination. Should such an outcome be deemed faulty or acceptable? In any case, due to inherently noisy sensing and actuation, the robot is highly unlikely to reach the exact intended position or to determine whether that position has been reached. Both P4 and P6 find specifying automated oracles challenging.

Furthermore, as explained by P4, in some cases, collecting data for the ground truth is either impossible or extremely expensive. P4 provided an example where a camera responsible for measuring the relative motion between two vehicles was under test. To validate the correctness of data provided by this camera, they needed a second method of measuring relative motion between the vehicles to act as the ground truth. The equipment and setup required to reach this ground truth turned out to be extremely expensive.

Following challenges described in C1, the vast space of inputs and corner cases makes it difficult to cleanly discriminate between correct-but-unusual and incorrect behaviors. P12 described this difficulty of defining suitable oracles for automated testing as "difficult to differentiate between bad behavior and correct, but strange behavior that is produced by unexpected inputs".

(C8) Software and hardware integration Robotic systems consist of software and hardware components [89, 176]. When asked about the most important feature of robotic systems that complicates testing, P1 responded with "robotic hardware". P2 said:

> *Robotics as a field is all about integration. Robotics is where hardware, software, and the world come together.*

To better understand the differences of a system with and without hardware components, let us present a quote from P9:

> *At the full hardware level, we see hardware that is flaky, like not assembling the cooling properly. In parts of robotics, you are writing multiple pieces of software, and you are running on specialized hardware, which might be optimized for performance, so there are extra concerns beyond traditional testing practices.*

The integration of components into a system can create unique testing challenges. P8 shared that even when software and hardware parts work properly in isolation, they frequently break once the software runs on physical hardware. In P9's opinion, developing a robotic system resembles developing many software and hardware systems all together (e.g., sensing, planning, and manipulation), and the simplest robot is at least three subsystems. Even though these subsystems work in isolation, unexpected failure modes are observed when they are combined. P3, P6, and P10 all faced confusion and challenges while running tests after integration of software and hardware components. P12 provided an interesting example of being limited by the battery on the robot after integrating software and hardware but not needing to worry about such problems when solely testing the software.

(C9) Distrust of simulation Simulation-based testing appears to be a promising approach to the challenge of test automation within the field of robotics [248, 291]. In the absence of simulation, full-system tests need to be executed on the real-world hardware in a real, physical environment, which significantly constrains the possibilities for test automation (e.g., regulations applied to testing autonomous vehicles on public roads).

Despite being aware of the theoretical value in using simulation to automate parts of the system testing process, many of our participants reported that they distrust the accuracy and validity of simulated operations. P2, P4, P8, P10, and P12 all believe the fidelity of simulation is not sufficiently high for testing and that running tests on the actual robot is the only way to test the system. For example, P2 said:

> *We mostly do field testing. That's what really affects what happens. The robot gets lots of impact from the environment. Simulation just doesn't reflect the real world.*

The lack of trust in synthetic results discourages developers from using software-based simulation as part of their test automation. In part, this could stem from the perceptions that our participants shared with us that many simulation tools are difficult to use and are more hassle than they are worth. For example, P10 mentioned that they could extend the simulator to be more faithful to the real world, but it is not worth the amount of time and effort to do that when they can just test on the real robot. P8 said:

> *I have more bodies that can test the hardware. I don't have time to build a Gazebo [plugin]. Getting the cameras to work properly in simulation is difficult.*

Table 3.5: For each theme, we indicate the challenges that support that theme.

ID	Title	Real-world complexities	Community & standards	Component integration
C1	Unpredictable corner cases	•		
C2	Engineering complexity	•		•
C3	Culture of testing		•	
C4	Coordination, collaboration, and documentation		•	
C5	Cost and resources	•	•	•
C6	Environmental complexity	•		
C7	Lack of oracle	•		•
C8	Software and hardware integration			•
C9	Distrust of simulation	•		

However, in some areas of robotics, notably self-driving cars, significant investments have been made in improving simulation [4, 8, 12]. P2 shared their opinion on this matter:

Robotics operates in such a variety of domains that developing high fidelity simulators is very difficult and for the most part do not exist today. However, if they did exist (and people trusted their fidelity) I think people would use them.

3.1.3 Interpretation and Discussion

In Section 3.1.2, we identified 12 testing practices used by robotics companies, and 9 challenges that roboticists face when designing, running, and automating tests. In this section, we identify 3 major themes among the identified challenges. We support these themes by showing quotes from our participants, and later speculate on the implications of each theme and provide suggestions for tackling their associated challenges. Table 3.5 describes the association between each of the three themes and the challenges of testing robotic systems that participants reported.

Real-world complexities By definition, robotic systems interact with the real world. This feature results in one of the major differences between robotic systems and traditional software systems. Interaction with complex, real-world environments is one of the most prominent challenges of testing robots that we observed in our interviews, and as P8 said:

Very little of the work on testing takes into account the physical aspects of the problem.

The complexities of the real world contribute to C1 and C2 as the large input space results in unpredictable corner cases and engineering complexity of specifying test inputs, and adds more complications to defining oracles discussed in C7. C6 and C9 are both impacted by real-world complexities as it is too difficult to make an abstraction of the environment, and testing in physical environments requires more resources (C5).

One way to attempt to simplify the complexities of the real world for the purposes of testing is simulation. However, developers still encounter many barriers when they attempt to use simulation [87]. As pointed out by our participants, simulators sometimes abstract away too many nuances of the real-world, and so developers do not feel comfortable relying on them. In other

cases, our participants responded that they feel that simulators are excessively complex to deploy, and since simulators often do not provide tools to manage that complexity, developers choose not to use them.

Other techniques that may be brought to bear on the challenges that arise from the interaction between robotic systems and the real world include *record and replay*, *model checking*, and *formal specification* [81, 100, 101, 111, 133, 162, 290]. As mentioned in C6, record and replay is a popular approach for dealing with the challenges of testing systems with complex environments. However, it is only a partial solution, because of its non-interactive nature. Model checking and formal specification are other solutions proposed to decrease reliance on simulators for automated testing by abstracting away the complexities of the real-world [77, 78, 91, 135, 183, 264, 296]. However, these systems are limited to specific types of systems, and, based on our study, have not yet been generally adopted in practice.

Furthermore, devising suitable oracles for full-system testing can quickly become an overwhelming task, as described in C7. To test their system, a developer may need to provide an oracle for several interrelated subsystems, all of which provide complex data. This can quickly become an overwhelming challenge. We believe that addressing this challenge of defining oracles for robotic systems requires the development of novel methods and techniques by the software engineering and robotics communities. In recent years, a number of studies have taken important steps towards tackling this problem [65, 124, 136, 288].

As a way of approaching the oracle problem for CPSs, studies have used metamorphic testing to observe the relations between the inputs and outputs of multiple executions of a CPS [179, 264, 294]. Even though metamorphic testing is a promising approach towards the oracle problem for cyberphysical and robotic systems, it requires identifying and proving metamorphic relations in the system [63].

We believe that the design and development of higher-fidelity simulators with better user interfaces and APIs may lead to a wider adoption of automated simulation testing. However, in absence of such simulators, the research community should develop novel tools and techniques for achieving test automation.

Community and standards Not all barriers to testing robotic systems are a result of technical issues. Another important theme of challenges we encountered are challenges that stem from community and standards. From our study, we learned that the robotics community is diverse and that people from different backgrounds may value testing and validation differently (C3). Many robotics practitioners are not familiar with methods of software testing (e.g., robustness testing and performance testing), and need guidelines to assist them in deploying testing practices (C4). With notable exceptions (e.g., industrial automation), the robotics industry is relatively young and immature, and the value of being the first to the market often outvalues the safety and quality of the system (C5).

Standards create an advantage for robotics companies by approving the product quality to the customers, and increasing the business value of testing. In our study, we found that robotics companies sometimes have standards from other industries that they can apply to certain domain-specific parts of their system, such as IEC standards from the vacuum industry for how many particles a vacuum should pick up [6]. However, for the robotics part of the system, there are

often no standards or guidelines. A number of standards have already been introduced for subdomains of robotics such as self-driving cars [84, 135], taking steps towards the right direction. However, we believe that more general-purpose guidelines and standards should be implemented to guide robotics developers, similar to those provided to other industries by UL or ISO [7, 11].

Component integration The third factor that complicates testing in robotic systems is the challenge of testing integrated hardware and software systems. In addition to C8, which is directly associated with this theme, integration of components contributes to C2, C5, and C7 as it increases the complexity of the system, the cost associated with testing, and introduces complications when defining oracles.

Some of the hardware and software challenges are similar to those found in embedded systems: timing, power consumption, memory allocation, and architecture [169]. However, in comparison to embedded systems, robotics hardware is often much more expensive and complex.

In many cases, the considerable expense of manufacturing robotic systems can limit the availability of hardware for testing. While embedded systems are often small, low-power devices with a fixed form and function, robots are often more of an extendible platform upon which physical components (i.e., sensors and actuators) can be added and removed over time. We believe that the development of tools and practices for testing robots in a more controlled fashion (e.g., hardware-in-the-loop testing [92]) may reduce the costs and risks associated with field testing on expensive hardware.

We believe that these three themes best describe the major challenges of robotics testing. Although partial solutions exist for some of these challenges[65, 124, 207, 221, 250, 289], in theory, the applicability and effectiveness of those solutions in practice remains unstudied and unclear. We observed that testing practices that are not represented by these themes, such as unit testing, continuous integration, and plan-based testing, were adopted by most of our participants. We also observe that the level of associated tooling and support for a given practice influences the uptake of that practice among robotics developers. For example, continuous integration is a practice that is well supported by tools and has been adopted by many of our participants. Logging and playback is also extremely popular among our participants. One reason behind this popularity could be the well-established tools and support around them, even though logging and playback still face challenges such as C1. In contrast, simulation testing and robustness testing are rarely adopted by our participants, as supporting tools and infrastructure have not been properly established yet.

3.1.4 Threats to Validity

Replicability *Can others replicate our results?* In general, qualitative studies can be difficult to replicate. We address this threat by making our interview script available on our companion site.[2] We cannot publish the interview transcripts because we promised our subjects that we would preserve their anonymity.

[2] `https://doi.org/10.5281/zenodo.3625199`

Construct *Are we asking the right question?* We used semi-structured interviews [251] to explore themes while also letting participants bring up new ideas throughout the interview. By allowing participants the freedom to bring up topics, we avoid biasing the interviews with our preconceived understanding of testing in robotics.

Internal *Did we skew the accuracy of our results with how we collected and analyzed the information?* Interviews can be affected by intentional or unintentional bias. To mitigate this concern, we followed established guidelines from literature [247], both designing and performing our interviews. Additionally, we ran a series of iterative pilots with robotics engineers, which we did not consider as data for the purposes of this work, but helped us shape a productive interview.

External *Do our results generalize?* Because there is a lot that is not known about testing in robotics, in this work we decided to prioritize depth over breadth. While our interviews did generate very rich data for us to analyze, we cannot make broad claims about how prevalent these practices are across the industry. To mitigate this threat, we constructed a sample with a specific eye for breadth, interviewing participants across a wide range of companies, sizes, and sectors.

3.2 Potential of Software-based Simulation for Testing

As described in Section 3.1, robotics practitioners generally distrust low fidelity simulation, and find it ineffective in exposing robotics bugs. Simulation, by necessity, represents a simplified abstraction of the environment and the system, and the robotics practitioners seem to believe simulation is not sufficiently expressive to trigger and support detection of bugs that manifest in reality.

To evaluate the validity of this common belief, it is necessary to first understand the factors, if any, that make reproducing and detecting bugs in simulation difficult. In a case study on ARDUPILOT system, my collaborators and I systematically produced a dataset of historical bugs, specifically seeking insight into the difficulties that underlie robotics testing in both simulation and deployment. We then characterized those defects to produce insights into the challenges and opportunities afforded by system-level test generation for such systems in simulation. An in-depth and nuanced knowledge of existing robotics bugs can lead to the development of techniques capable of catching future bugs with similar characteristics. Moreover, if a tool can detect bugs that humans have previously identified in these systems, such a tool may also be able to detect other latent bugs in the system, especially if the latent bugs have similar characteristics to earlier-identified bugs.

Although datasets of robotics bugs do exist [72, 114, 254, 257, 276], none allows faults to be reproduced and inspected in simulation, nor do their accompanying analyses investigate the difficulties of triggering and detecting bugs. Ensuring that bugs can be reliably reproduced allows datasets to be used for a rich diversity of studies, including testing, fault localisation, and automated program repair, as similar datasets for non-robotic systems [80, 126, 145, 168, 242, 260] have demonstrated in broader contexts. These studies inspire our work to recreate and detect robotics and autonomous systems bugs in simulation, with a view towards detecting new bugs, which is a direction the previous work does not take. Indeed, well-defined benchmarks and datasets can be instrumental for clarifying and advancing a coherent definition of a discipline's dominant research paradigms [252].

This study is published in the International Conference on Software Testing, Verification and Validation (ICST 2018) [265].

3.2.1 Methodology

In this section, I first discuss our process for identifying bug-fixing commits within the version-control history of the ARDUPILOT project. I then discuss how we transform each bug-fixing commit into an executable Docker image, capable of reproducing the bug in simulation. Finally, I describe how we analyzed each bug.

Bug Collection To identify which of the $> 29,000$ commits within the version-control history of the ARDUPILOT represent bug fixes, we used GITPYTHON[3] to mine potential bug-fixing commits from the project's GITHUB repository[4]. To do so, we implemented a script that uses a multi-stage process to identify commits.

1. To ensure reproducibility, we restricted our attention to all $29,081$ commits within the repository that occurred before October 1st, 2017.
2. Next, we removed all commits that do not modify at least one `.cpp`, `.hpp`, or `.pde` file. We ended up with $24,897$ commits after this step.
3. We then filtered the set of commits to those whose descriptions contain either of the following terms: "bug" or "fix". There were $2,213$ commits with these keywords in their description. From manually trawling the commit history, we found that the majority of bug-fixing commits use at least one of these terms.
4. We then focused our attention on commits related to the ARDUPILOT's vehicle controller modules, with the exception of the ARDUSUB modules, since at the time, there existed no simulator. To identify commits related to these modules, we exploited ARDUPILOT's conventions for writing commit descriptions[5] to determine the module that was modified by the commit. After this stage, 414 commits remained.
5. Finally, we performed another round of keyword filtering, to drop all commits containing a taboo term, suggesting that the bug is not relevant to our dataset. In this stage, we remove commits we believed to be related to the build system, compilation, documentation, or cosmetic changes. A complete set of keywords, can be found in the script (included as part of our dataset). At the end, we found 333 commits that satisfied all the filters.

After automatically identifying the likely bug-fixing commits using our script, we manually inspected each commit, and discarded those that we deemed to be irrelevant to our dataset. We deemed refactorings, compilation bugs, cosmetic tweaks, and documentation changes to be irrelevant. We also excluded commits that we deemed to be improvements; this included both non-functional improvements (e.g., use of a more memory-efficient algorithm, `6da68c53`), and

[3] `https://github.com/gitpython-developers/GitPython`

[4] `https://github.com/ArduPilot/ardupilot`

[5] Since April, 2013, almost all commits to the ARDUPILOT repository observe the following form: "*submodule*: *description*", where *submodule* describes that submodule that is modified by the commit, and *description* provides a description of the changes.

functional improvements (e.g., "nose of copter now points at next guided point when it is more than 10m away", `0460147a`).

To reduce the likelihood of falsely including or excluding a bug from our dataset, two persons of our group, both of whom are familiar with the implementation of the ARDUPILOT platform, independently marked each commit as relevant or irrelevant; in the case that the reviewer determined the commit to be irrelevant, a reason was provided. Following this process, the reviewers unanimously agreed to remove 63 commits, and disagreed over the relevancy of 57 commits.

To settle the disputed commits, an independent party served as an arbiter, and was given the responsibility of determining the relevancy of the commit. The arbiter deemed 42 of the 57 disputed commits to be irrelevant. In total, we identified 228 commits as bug fixes. Our approach to independent bug classification is similar to that used in previous work [193].

Packaging After identifying the set of suitable bug-fixing commits in the version control history of the ARDUPILOT project, we set about packaging them into minimal Docker containers[6], capable of consistently reproducing the bug within the confines of simulation. To reproduce the bug, we followed an approach similar to that used by MANYBUGS [168], a widely used of historical bugs in large-scale C programs, by using the version (i.e., commit) of the source code immediately before the bug-fixing commit.

We excluded ten of the commits from the dataset, but included them in our analysis, since they only manifest when executed on specific physical hardware (e.g., "fix LED notify during auto esc calibration"; `a3450a95`).

We have freely released the Dockerfiles used to construct the images for each bug, together with the results of our analysis, and scripts for reproducing our bug collection process.[7] Prebuilt container images may also be downloaded from DockerHub, as described in the released artifacts.

Characterization After reaching a consensus on the list of bug-fixing commits, we manually inspected each commit to determine whether the bug can be reproduced in simulation, and if so, what are the requirements for *triggering* and *detecting* it. To obtain this information, we answered the following questions:

1. **Does triggering or observing the bug rely on physical hardware?** We ask this question to determine whether software-in-the-loop simulation approaches are sufficiently capable of detecting most bugs, or whether the majority of bugs require physical hardware.
2. **Is the bug only triggered when handling concurrent events?** Parallelism and concurrent events are inherent features of most robotics systems due to their physical nature. Bugs of this nature cannot be triggered by subjecting the system to a sequential stream of commands. Automatic detection of such bugs may prove especially challenging; specification languages, such as process calculi [36] and timed automata [44], are required to describe how the system should behave under such circumstances.

[6] https://www.docker.com/

[7] https://github.com/squaresLab/ArduBugs

We ask this question to determine how many bugs can be triggered and detecting using simpler modeling approaches, restricted to describing the behavior of the system in response to a sequential stream of commands.

3. **Which kinds of input are required to trigger the bug?** In most cases, inputs are essential for triggering the bug. ARDUPILOT allows inputs to be provided to the system in a number of different ways. Discrete inputs, such as preprogrammed missions and ground control commands, are more amenable to automated testing than continuous inputs, such as radio-control inputs. Bugs that require more than one kind of input place an even greater strain on testing techniques. We ask this question to determine how many bugs can be triggered using only a single discrete input type.
4. **At which stage in the execution does the bug manifest?** Bugs can occur at different points during execution; they may manifest during initialization or normal operation, or they may occur during failure recovery and system reboot. Handling failure recovery and system reboots places additional requirements on testing approaches, and requires that failure conditions can be triggered. We ask this question to determine how many bugs can be triggered without the need to induce failure or a system reboot.
5. **Is the bug only triggered under certain configurations?**

   Bugs that depend on the *static* configuration of the system only manifest when the system is compiled with certain options. Similarly, bugs may only trigger under certain *dynamic* configurations of the system (i.e., parameters supplied to the system at run-time). We ask this question to determine how many bugs are only triggered under certain static and/or dynamic configurations.
6. **Is the bug only triggered in the presence of certain environmental factors?** Environmental factors include the presence of obstacles and geographical features (e.g., hills and valleys), wind and weather conditions, unreliable sensor behavior, and the need for human interaction. Testing bugs that require such triggers places an additional burden on the simulation environment, and vastly increases the search space. We ask this question to determine how many bugs can be triggered without requiring specific environmental conditions.
7. **How does the bug affect the behavior of the system?** Bugs can manifest in a diversity of ways, and have varying consequences on the reliable operation of the system. Characterizing the exact effects of each bug is difficult, error prone, and hard to interpret. To that end, we broadly classified the effects of each bug as either *logging-related*, *behavioral*, or *(program) crashing*.

   As their name suggests, crashing bugs are known to cause the program to crash. Logging-related bugs corrupt the log files or cause incorrect status messages to be produced, but do not otherwise affect the run-time behavior of the robot. Behavioral bugs manifest in observable changes to the behavior of the robot; for these bugs, we also provided a qualitative description of how the behavior of the robot is affected.

The inspection process consisted of reading the commit description, understanding the effects of the changes made by the fix, and in some cases, executing the buggy version. To reduce the likelihood of a false label, two of the authors independently went through the list of bugs and

assigned labels. The authors disagreed on more than 150 out of approximately 1,600 labels; in those cases, an independent arbiter made the final decision.

3.2.2 Results

In this section, I present the results of our analysis on the set of bugs found in ARDUPILOT code repository. In total, we collected 228 bugs in three ARDUPILOT subsystems: 157 for ARDUCOPTER, 50 for ARDUPLANE, and 21 for ARDUROVER.

Fix Characteristics The median number of files changed by the identified bug-fixing commits is one, and the median number of line insertions and deletions, according to `git diff`, is five. 183 of the bugs were fixed by modifying a single file; this finding is encouraging for the prospects of performing fault localization and program repair in robotics systems. The true number of line insertions and deletions related to the bug fix is likely to be lower than the observed median; a number of the commits perform extensive refactoring, unrelated to the bug.

Bug Characteristics Studying the characteristics of bugs can provide us with the insight of the nature of bugs in the system and help us to improve bug detection and test-generation techniques. Below, we discuss the findings of our analysis in terms of the questions proposed in Section 3.2.1.

1. **Does triggering or observing the bug rely on physical hardware?**

   In total, only 10 of 228 bugs relied on the presence of physical hardware for their detection or observation. 5 of the 10 bugs concerned platform-specific code for the robot, and thus cannot be tested using software-in-the-loop simulation. 4 of the 10 bugs affected the robot's lights and sounds; in theory, these bugs may be detected by using a higher-fidelity simulator. The remaining bug (`52c4715c`) only manifested on hardware with low memory capacity.

   This finding suggests that software-in-the-loop simulation approaches are capable of detecting the majority of bugs within robotics systems. By alleviating the need for specific physical hardware, the time and cost of testing robotics systems can be reduced by using cloud-computing resources.

2. **Is the bug only triggered when handling concurrent events?**

   To our surprise, we determined that only 13 out of 228 bugs (5%) require concurrent events in order to be triggered. This particularly interesting result demonstrates that automated testing techniques with simple oracles, capable only of capturing the expected behavior of sequential streams of events, may be sufficient to detect the majority of bugs in robotics systems. We believe that the lack of a need to describe parallel behaviors of the system reduces the specification burden on designers, and thus increases the likelihood of the acceptance of automated testing techniques.

3. **Which kinds of inputs are required to trigger the bug?**

   We found that 9 bugs can only be triggered by the system's command-line interface, which is used to interact with the robot when it is docked and tethered. The other 219 bugs are triggered by ground control system (GCS) commands, preprogrammed missions, and/or

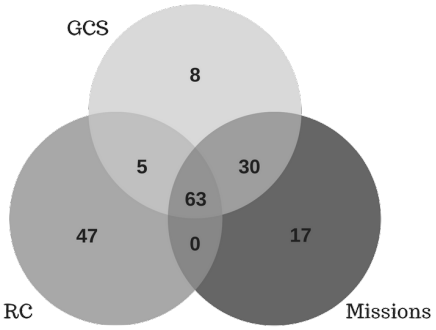


Figure 3.1: The number of bugs that could be triggered by exclusively using each input type. The intersection of input types shows the number of bugs that can be triggered by any of them. Less than a quarter of the bugs rely on continuous radio-controller inputs, which are normally provided by a human-operator.

RC inputs. Only 7 of those 219 bugs relied on more than input type in order to be triggered (e.g., GCS commands and RC inputs). Figure 3.1 illustrates how many of the remaining 212 bugs could be triggered by each non-CLI input type.

Mimicking continuous radio-controller (RC) inputs, usually provided by a human-operator, is significantly more challenging than supplying the system with discrete, well-formed inputs (i.e., GCS commands, and preprogrammed missions). Encouragingly, our findings show that 165 of the 212 bugs do not rely on continuous input. Just under half of the bugs (106) can be triggered by exclusively using GCS commands, which place the least requirements on the oracle.

4. **At which stage in the execution does the bug manifest?**

   We discovered that 20 bugs occur when the system is in its preflight phase (i.e., initialization), 10 manifest during its failsafe and recovery behaviors, 3 happen following a soft reboot, and 11 are encountered during the tuning phase. The remaining 184 bugs are triggered during the normal operation of the robot. Some of these execution stages are more difficult to simulate, model, and encounter. For example, testing the failsafe behavior of the robot requires that failures can be induced during the simulation.

   Importantly, the observation that almost 80% of bugs occur during the normal operation of the robot shows that automated testing techniques do not necessarily need to cover all modes of operations in order to detect the majority of bugs.

5. **Is the bug only triggered under certain configurations?**

   We determined that 81 of 228 bugs depend on either a particular static (53) or dynamic (20) configuration, or a combination of both (8). Knowing that more than one-third of bugs depend on a particular configuration to be triggered demonstrates the importance of testing the system under a wide range of configurations. It may be fruitful for automated testing techniques to explore the configuration space.

6. **Is the bug only triggered in the presence of certain environmental factors?**

To our surprise, once again, we discovered that only 22 of 228 bugs depend on environmental factors. For example, `36634265` is only encountered in windy conditions. Other bugs, `1e2e24ee` and `436062ef`, require a human to be physically present in order to throw the plane. These bugs can be especially challenging to detect due to the demands they place on the fidelity of the simulation, and the need to test the system against a large and variegated range of environments.

Crucially, this finding shows that simpler (and implicitly more efficient) automated testing techniques that do not attempt to account for environmental factors are capable of triggering the majority of bugs.

7. **How does the bug affect the behavior of the system?**

   We found that 17 bugs only cause corruption of the log files, or report incorrect status messages to the user. From inspecting the commit message, the modifications to the source code, and associated bug report, where one was provided, we determined that 6 bugs were reported to or are likely to crash. The remaining 205 bugs resulted in observable, behavioral changes to the program. Descriptions of the effects of these bugs are provided as part of our dataset.

   These findings suggest that techniques such as fuzz testing are unlikely to detect many bugs. The vast majority of bugs can be detected by solely observing the run-time behavior of the robot. Although log-checking-based techniques exclusively detect a relatively small number of bugs, incorporating log information into the oracle could allow certain behavioral bugs to be detected more easily.

3.2.3 Threats to Validity

There is a risk that our results may not generalize beyond the single system that we used as our case study. Our study goals motivated us to focus on a single system in depth, rather than performing a (necessarily) less-detailed study of multiple systems. This risk is mitigated by the rising popularity of ROS-based and Ardu* systems in the consumer market, increasing the potential utility of our results even if they do not fully generalize. While we do not have exact numbers for the number of Ardu* systems' users, we note its active GitHub development with 350 contributors and over 40,000 commits. It is also possible that the predominance of simple bugs in our dataset was because it was easier for users to report those bugs and for developers to fix them. However, the fact that the software enjoys continued use and popularity suggests that the developers have fixed the the bugs that most interfered with use. Our findings are also corroborated by similar studies [72, 114], which found, for example, that relatively simple bugs predominate even in complex software. The bugs we studied were drawn from all commits; it was unclear for many whether they were discovered in the field or in simulation. Additionally, the system we studied operates on a relatively simple control loop design; systems with more complex architectures may have bugs that are less amenable to being replicated in simulation. Addressing this risk motivates future work characterizing defects in more complex systems.

Our approach to identifying bugs is neither sound nor complete – a known risk in developing datasets from source control histories [51]. Some bug-fixing commits do not satisfy our search criteria, and so they are excluded from the dataset. Additionally, it may be the case that certain

kinds of bugs are more likely to satisfy our criteria, leading to an unrepresentative dataset. This risk is likely *more* applicable to non-cyberphysical systems, where critical live bugs can be more easily found and fixed over the course of normal development.

Similarly, there is a risk that commits were incorrectly labeled as bug fixes and non-bug fixes during the manual phase of the bug identification process. To mitigate this, we used a voting system to perform identification. Indeed, although we made our best efforts to usefully label the dataset, we may have mislabeled portions of the dataset or not chosen the most useful labels. Our approach to using consensus to classify commits as bug fixes is similar to that used by Martinez and Monperrus [193] in their work on learning the shapes of bug-fixing patches. However, note that the previous work had all three examiners inspect each commit; we only require that a third examiner inspect a commit if the other two examiners disagree or are unsure. We were unable to characterize 10 commits, which we subsequently dropped from the dataset. We mitigate this threat by performing three passes through the dataset and having several evaluators adjudicate disagreements, and by releasing our dataset and analysis results publicly for replication and review by other researchers.[8]

Conclusion The findings of our study strongly support the idea of applying cheap, simulated-based testing approaches to the problem of detecting bugs in robotics systems. However, we also found that continuous events, in the form of radio-controller inputs, and specific configurations are required to trigger a large number of bugs. We believe that both of these challenges, whilst difficult, can be overcome by developing specialized testing methods and leveraging and building upon existing knowledge in, e.g., testing of highly configurable systems [151].

3.3 Challenges of Using Simulators for Test Automation

Our case study on ARDUPILOT system described in Section 3.2 showed that low-fidelity SITL simulation can be used for systematic, large-scale automated testing of CPSs to discover a large number of bugs in the system, and can take us one step closer to the automated testing pipeline in Figure 1.1.

However, in the qualitative study presented in Section 3.1, our participants referred to a number of challenges of using software-based simulators, and discussed them in depth. These challenges included the low-fidelity of simulators, their hard to use interface, and the cost and effort required to set them up for a particular system.

Given the considerable potential of simulation-based testing, my collaborators and I set out to understand the extent to which simulation is used for testing in practice, and identify the barriers that prevent wider use. Prior studies have examined technical features of particular robotics simulators [75, 228, 255] but paid little attention to their role in quality assurance or the challenges developers face when using them. Instead, prior work on the challenges of testing in robotics [20, 183] and CPSs in general [292] broadly identifies simulation as a key element of testing that requires improvement.

[8] https://github.com/squaresLab/ArduBugs

We conducted a study of robotics developers to understand how they perceive simulation-based testing and what challenges they face when using simulators. Through a survey of 82 robotics developers, we found that *simulation is used extensively for manual testing,* especially during early stages of design and development, but that *simulation is rarely used for automated testing.* By analyzing participant responses using grounded theory and other qualitative and quantitative methods, we identified 10 challenges that make it difficult for developers to use simulation in general (i.e., for any purpose), for testing, and specifically for automated testing. The challenges include a lack of realism, a lack of reproducibility, and the absence of automation features.

The results of this study can inform the construction of a new generation of software-based simulators, designed to better accommodate developers' needs for robotics testing. In particular, we show how several key barriers that impede or prevent simulation-based testing are incidental software engineering challenges, such as the need for languages and tools to construct test scenarios and environments. The software engineering and testing communities are well positioned to study and address these challenges of testability. This study is published in the International Conference on Software Testing, Verification and Validation (ICST 2021) [21].

3.3.1 Methodology

We aimed to better understand the ways in which robotics developers use simulation as part of testing processes, and the challenges they face in doing so. Here I describe our methodology by presenting our research questions, survey design, participant recruitment, and analysis methods. Finally, I discuss some of the threats to the validity of our study and their mitigation.

Research Questions To assess the ways in which robotics developers use simulation for testing, we first ask the following research question:

- **RQ1:** *To what extent do developers use simulation for testing and test automation?*

Following the first research question, we focused on identifying the challenges robotics developers face when using simulation for testing. These challenges consist of both general limitations that impact all use-cases of simulation and challenges that specifically affect testing and test automation. Making the distinction between general limitations of simulators and testing-related challenges allows us to better understand and illustrate the issues that if resolved, can result in higher adoption of simulation for testing. As a result, we categorized the challenges of using simulation in three groups – general, testing-specific, and test-automation-specific challenges – in the following research questions:

- **RQ2:** *What challenges do developers face when using simulation in **general**?*
- **RQ3:** *What challenges do developers face when using simulation for **testing**?*
- **RQ4:** *What challenges do developers face when using simulation for **test automation**?*

Survey Design To answer our research questions, we conducted an online survey of robotics developers in November 2019. We followed best practices in survey design by explicitly breaking down the research questions into targeted survey questions, creating and pre-testing a pilot survey

Table 3.6: Examples of survey questions separated by their corresponding research question (RQ). The full list of questions can be found at https://doi.org/10.5281/zenodo.4444256.

RQ1	• Have you ever used a software-based simulator? • For what purposes have you used software-based simulation?
RQ2	• Please tell us about how you used software-based simulation in [your latest] project? • Have you ever decided not to use software-based simulation for a project?
RQ3	• Which of these features are most useful when you use software-based simulation, specifically for testing?
RQ4	• How did you use software-based simulation as part of your test automation? • For what reasons, if any, have you not chosen to attempt to use software-based simulation for (partially) automated testing?

on a representative population of sample respondents, and making adjustments based on feedback until reaching saturation [68, 157, 235]. Examples of the survey questions are presented in Table 3.6.

To ensure a meaningful interpretation of results, we provided our participants with a definition for the term "testing" as "any approach to finding errors in any part of a system (e.g., the software or hardware) by executing code used in the system in whole or in part, which can take place at any stage of system development and may be either automated or manual." Note that this definition is intentionally broader than that used in our previous study on the challenges of testing robotic systems (Section 3.1), generally, which did not consider the manual use of simulation during the early stages of development to be a form of testing [20]. The full list of our survey questions, together with terminology and examples, are provided as part of the supplementary materials at https://doi.org/10.5281/zenodo.4444256.

Recruitment To reach our intended audience (i.e., robotics developers), we distributed our survey via social media outlets, email, and several popular forums within the robotics community: the ROS and Robotics subreddits on Reddit,[9] the ROS Discourse,[10] and the RoboCup forums.[11] We decided to advertise our survey to the ROS community as ROS is a popular and widely used robotics software framework [229, 279]. We also advertised our survey on Facebook and Twitter and posted a recruitment email to mailing lists for a university robotics department and a robotics research institution.

In total, 151 participants took the survey, out of which 82 completed it. For the purpose of analysis, we only consider the 82 completed responses. All 82 participants who completed the survey reported that they had used a robotics simulator. Table 3.7 presents the demographics of these 82 participants. In terms of experience, more than two thirds of participants (71.95%)

[9] https://reddit.com
[10] https://discourse.ros.org
[11] http://lists.robocup.org/cgi-bin/mailman/listinfo

Table 3.7: Demographics for the 82 survey participants that completed the survey in terms of their experience, the types of organization at which they had worked, and the size of the most recent organization to which they belonged.

Experience			**Organization**			**Size of organization**		
Years of experience	#	%	Type	#	%	Number of people	#	%
Less than one year	10	12.20%	Academia	65	79.27%	1–10 people	22	26.83%
Between one and three years	13	15.85%	Industry	54	65.85%	11–50 people	23	28.05%
Between three and ten years	40	48.78%	Individual	35	42.68%	51–100 people	9	10.98%
More than ten years	19	23.17%	Government	12	14.63%	>100 people	28	34.15%
			Other	9	10.98%			

reported having worked with robotics software for more than three years. Most participants (79.27%) reported that they had worked with robotics in academia at some point during their life, and almost two thirds (65.85%) reported working with robotics in industry at some point. Participants reported that they currently work at organizations of varying sizes. Overall, our study sample is composed of a diverse array of candidates with differing levels of experience who have worked in a variety of organizations, thus ensuring that the results of the study are not limited to any one population.

Analysis Our survey includes both quantitative (closed-ended) and qualitative (open-ended) questions. To analyze the open-ended responses, we used descriptive coding [244] to assign one or more short labels, known as *codes*, to each data segment (i.e., a participant's response to a given question), identifying the topic(s) of that segment. After developing an initial set of codes, we adjudicated to reach consistency and agreement, then used code mapping to organize the codes into larger categories [60, 186, 244]. Using the explicit mapping from survey questions to research questions, devised during survey design, we aggregated the set of relevant categories for each research question. Finally, we used axial coding to examine relationships among categories and identify a small number of overarching themes for each research question.

Threats to Validity To mitigate the threat of asking the wrong questions and introducing bias in the wording of questions, we followed survey design best practices [68, 157, 235], such as the use of iterative pilots, and included a number of open-ended questions to allow participants to freely discuss topics of concern to them.

Our analysis of open-ended survey responses is a potential threat to internal validity. To mitigate this concern, we followed established guidelines on qualitative and quantitative studies [60, 186, 244]. As a post-study validation, we shared the results and conclusions on several public platforms, including those used for recruitment, and received positive feedback on our findings from the robotics community.

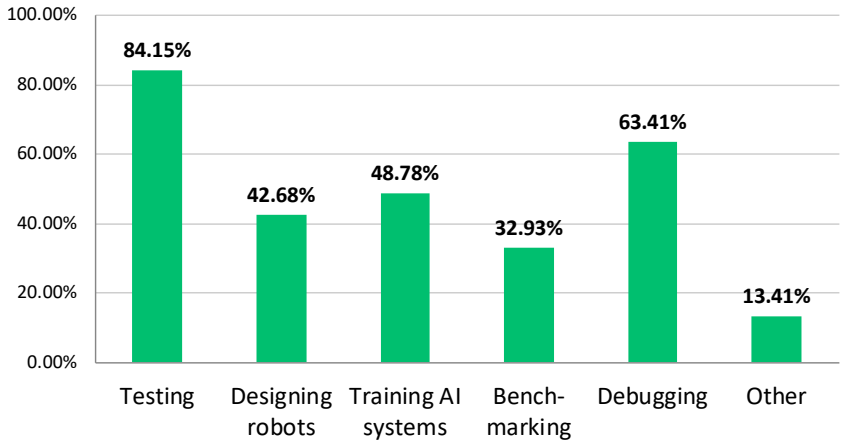


Figure 3.2: An overview of the high-level reasons that participants gave for using simulation (82 responses).

Although we received many responses from robotics developers, we cannot make broad claims on the generalizability of our findings. To mitigate this threat, we distributed the survey among robotics developers from different backgrounds and organizations by targeting popular robotics platforms.

To promote further research, we share our recruitment materials, questionnaire, codebook, and additional results at the following URL: `https://doi.org/10.5281/zenodo.4444256`.

3.3.2 Results

In this section, I present the results of our study of robotics developers on their use of simulation for testing, and the challenges they face along the way. The full list of identified challenges are presented in Table 3.8.

RQ1: To what extent do developers use simulation for testing and test automation? Our survey asked participants about their use of simulation: both broadly and specifically for testing robotics systems. We find that our participants are unanimously familiar with simulation, and they use it on a regular basis. 59 out of 82 (71.95%) participants reported that they used simulation within the last month at the time of completing the survey. When asked about their most recent project that involved simulation, 51 of 82 (62.20%) participants reported that they used a simulator daily, and 25 of 82 (30.49%) participants reported that they used a simulator on a weekly basis.

Figure 3.2 presents the variety and popularity of purposes for which our participants use simulation. Almost 84% of participants have used simulation for testing, and testing is the most popular use case for simulation. This suggests that developers generally see value in using simulation for testing.

Participants reported using simulation for various forms of testing, including: testing the underlying algorithms; variability testing (e.g., testing the robot with different components);

Table 3.8: Summary of challenges participants encountered when using simulation in general (■), specifically for testing (▲), and for test automation (★).

	Challenge	Description
■	*Reality gap*	The simulator does not sufficiently replicate the real-world behavior of the robot to a degree that is useful.
■	*Complexity*	The time and resources required to setup a sufficiently accurate, useful simulator could be better spent on other activities.
■	*Lacking capabilities*	Simulators may not possess all of the capabilities that users desire, or those simulators that do may be prohibitively expensive.
▲	*Reproducibility*	Simulations are non-deterministic, making it difficult to repeat simulations, recreate issues encountered in simulation or on real hardware, and track down problems.
▲	*Scenario and environment construction*	It is difficult to create the scenarios and environments required for testing the system in simulation.
▲	*Resource costs*	The computational overhead of simulation requires special hardware and computing resources which adds to the financial cost of testing.
★	*Automation features*	The simulator is not designed to be used for automated testing and does not allow headless, scripted or parallel execution.
★	*Continuous integration*	It is difficult to deploy the simulator in suitable environments for continuous integration (e.g., cloud computing servers).
★	*Simulator reliability*	The simulation is not reliable enough to be used in test automation in terms of the stability of the simulator software, and the timing and synchronization issues introduced by the simulator.
★	*Interface stability*	The simulator's interface is not stable enough or sufficiently well-documented to work with existing code or testing pipelines.

sanity checking (e.g., checking software in simulation before deploying it to the hardware); and multi-robot testing (e.g., simulating how robots will interact with each other).

Participants also reported a variety of reasons for using simulation for testing. These include when it is unsuitable or impractical to test on real hardware or in a real environment. They also reported using simulation to better understand the design and behavior of existing robotic systems and their associated software, and to incorporate simulation into automated robotics testing, including continuous integration (CI).

Of the 84% of participants who have used simulation for testing, we found that 61% of them have also tried to use simulation as part of their test automation. These findings demonstrate that developers find simulation to be a valuable tool for testing, and there is a desire to incorporate

Table 3.9: An overview of the reasons that participants gave for not using simulation for a particular project, based on 28 responses.

Reason for not using simulation	**#**	**%**
Lack of time or resources	15	53.57%
Not realistic/accurate enough	15	53.57%
Lack of expertise or knowledge on how to use software-based simulation	6	21.43%
There was no simulator for the robot	4	14.29%
Not applicable	4	14.29%
Too much time or compute resources	2	7.14%
Nobody suggested it	0	0.00%
Other	2	7.14%

simulation-based testing into their test automation processes.

These results motivated the rest of our study, which looks at the challenges robotics developers face using simulation. Given the ubiquity of simulation and its importance to robotics testing and development, there is great potential benefit from lowering the barriers to using simulation, especially for testing. We highlighted these barriers to direct attention to areas in which improvements to simulators may have the most impact, thereby allowing developers to advance the state of software engineering and quality assurance in robotics.

Our key insight is that simulation is an essential tool for developers that is used extensively for building and testing robot software. Given its importance, it is vital that we better understand the challenges that prevent developers from realizing its full potential.

RQ2: What challenges do developers face when using simulation *in general?* Although we found that simulation is popular among developers, 28 of 82 (34.15%) participants reported that there was a project for which they decided to not use simulation. Their reported reasons are given in Table 3.9. By analyzing these reasons along with the difficulties that participants experienced when they did use simulation, we identified three high-level challenges of using simulation in general, discussed below.

Reality gap Simulation, by definition, creates an abstraction of the real world and the robotics hardware in it. As a result, it can never be 100% accurate in representing all aspects of the real environment. The sometimes inadequate representation of physical reality in simulation is known colloquially as the *reality gap*. Many participants cited the reality gap both as a challenge when trying to use simulation and a reason not to use it in the first place. P33 notes that simulation can produce unrealistic behaviors that would not occur in the real world. P16 highlighted that accounting for all relevant physical phenomena can also be challenging: "my simple simulation model did not include a tire model, so simulations at higher speeds did not account for realistic behaviors for cornering or higher accelerations or deceleration." In particular, realistically modeling stochastic processes, such as signal noise, and integrating those models into the simulation as a whole is a challenge: P15 shared, "A classic problem is integrating wireless

network simulation with physical terrain simulation. This also applies to GPS signal simulation, as well."

For some, such as P29, the reality gap can be too large to make simulation valuable: "too big discrepancy between simulation results and reality (physical interaction)." For others, simulation can still serve as a valuable tool despite the existence of the reality gap. As P36 puts it, "Software behavior in simulation is different compared to [the] real [world], so not everything can be tested, but a lot can be." In talking about the reality gap, respondents faulted both the limitations of the modeling formats and the limitations of the simulators.

Complexity Accurate simulation of the physical world is inherently challenging and involves composing various models. Alongside the essential complexity of simulation are sources of *accidental complexity* [55] that do not relate to the fundamental challenges of simulation itself, but rather the engineering difficulties faced when trying to use simulation. For example, a lack of user-friendly features is a source of accidental complexity. Sources of accidental complexity may ultimately lead users to abandon or not use simulation at all.

Inaccurate, inadequate, or missing documentation can make it difficult to learn and use a simulator. P22 highlights that a "lack of documents for different platform types and sometimes wrong documentation makes us lose a lot of time working on [stuff] that will never work, for example, the Gazebo simulator does not work well in Windows." A language barrier may cause documentation to be inaccessible, as reported by P74: "The language was Japanese, but we don't speak that language so we couldn't use well the simulator."

Difficult-to-use API make it difficult to extend the simulator with new plugins. P4 points out that "Gazebo is the de-facto [simulator] right now and is poorly documented and difficult to customize to any degree." A lack of integration with popular computer aided design (CAD) software (e.g., AutoCAD, SolidWorks) and support for industry-standard 3D modeling formats (e.g., IFC), makes it difficult to import existing, high-quality models.

Together, these sources of complexity increase simulators' learning curve and may lead developers to abandon or never start to use them. P20 shared that there is a "steep learning curve in understanding the test environment software setup and libraries. Without a good software engineering skills the simulated environment will not replicate the real environment."

Lacking capabilities Finding a simulator that provides all of the characteristics a user desires can be challenging. P77 highlighted that, while it is possible to find a simulator that is good in one particular aspect, it is hard to find a simulator that is good in all desired aspects. As P4 pointed out, simulators that do possess all of the desired qualities tend to be expensive: "Adding plugins is usually very challenging, and the only good frameworks that do any of this stuff well are very expensive (V-Rep and Mujoco for example)."[12]

We asked which simulation features participants wanted most but are unable to use in their current setups. Among the most important features mentioned were the ability to simulate at faster-than-real-time speeds (i.e., where the simulation clock runs faster than the wall clock),

[12]Coppelia Robotics informed the authors that V-REP has been re-branded as CoppeliaSim and is free for educational and non-commercial applications.

native support for headless (i.e., without the GUI) execution, and an easier means of constructing environments and scenarios.

Numerous participants wanted the ability to run simulation at faster-than-real-time speeds but were unable to do so in their current simulation setups. For example, P52 said, "We needed to speed up simulation time, but that was difficult to achieve without breaking the stability of the physics engine." This feature is useful not only for reducing the wall-clock time taken to perform testing, but for other purposes, as P62 highlighted: "Faster than real time is really important to produce training data for deep learning."

Several participants also desired features that would increase simulation fidelity (i.e., how closely the simulation mimics reality). P46 wanted support for "advanced materials in environments (custom fluids, deformable containers, etc.)." Interestingly, P69 desired the ability to tune the fidelity of the simulation: "Ability for controllable physics fidelity. First order to prove concepts then higher fidelity for validation. Gazebo doesn't have that." Recent studies have shown that low-fidelity simulation can effectively and inexpensively discover many bugs in a resource-limited environment [234, 254, 265].

Participants also specified other capabilities such as native support for multi-robot simulation and large environments, and support for efficiently distributing simulation computations across multiple machines.

Ultimately, the complexities of setting up and using simulation, the reality gap, and the time and resources necessary to make the simulation useful led some participants to use physical hardware instead. As P4 said, "It was easier and more accurate to setup and test on a physical system than simulate."

Our key insight is that developers find considerable value in simulation, but difficulties of learning and using simulators, combined with a lack of realism and specific capabilities, constrain the way that developers use simulation. By alleviating these challenges, simulation can be used for a wider set of domains and applications.

RQ3: What challenges do developers face when using simulation for *testing*? Participants reported a variety of challenges in attempts to use simulation for testing, summarized in Table 3.8. We identified the following challenges that mainly affect the use of simulation for testing:

Reproducibility From inevitable sensor and actuator inaccuracies to stochastic algorithms (e.g., vision and navigation), robotics systems have numerous, inherent sources of nondeterminism. Temporal effects (e.g., message timing and ordering) can also lead to different outcomes. A deterministic simulator should be able to simulate nondeterministic factors but also allow reproducibility, in which it repeats a given simulation with all of these factors behaving in the same manner. For example, a deterministic simulator should be able to execute two simulations of the same scenario, producing the exact same messages in the exact same order. Such a simulator would allow for consistent testing and fault diagnosis.

The lack of reproducibility and presence of non-determinism in simulators lead to difficulties when testing, as reported by participants. P42 highlighted that a "Lack of deterministic execution of simulators leads to unrepeatable results." This points to a need to accurately reproduce system failures that are discovered in testing, in order to diagnose and debug those failures. If a tester

cannot consistently reproduce the failures detected in simulation, it will be difficult to know whether changes made to the code have fixed the problems. P7 pointed to the particular difficulty with achieving reproducibility in Gazebo: "Resetting gazebo simulations was not repeatable enough to get good data." P48 and P81 also mentioned a desire for reproducibility.

Consistent and systematic testing often relies on deterministic test outcomes, particularly when incorporating test automation and continuous integration tests, which rely on automatically detecting when a test has failed.Flaky [203] and non-deterministic tests may lead to a false conclusion that a problematic software change does not have a problem (a false negative) or that a good change has problems (a false positive).

Scenario and environment construction Testing in simulation requires a simulated environment and a test scenario, which can be understood as a set of instructions for the robot under test. Participants reported difficulty in constructing both environments and test scenarios. P38 said: "Setting up a simulation environment is too much work, so I don't do it often," and P3 contributed, "Scripting scenarios was not easy. Adding different robot dynamics was also not easy." They wanted to be able to construct these more easily or automatically. Participants pointed out that the scenarios or environments they require sometimes must be created "by hand," which requires a heavy time investment and is subject to inaccuracies. In recent years, high-level languages have been proposed to aid the construction of rich driving scenes in simulation [97, 158, 187]. P4 said, "Making URDF[13] files is a tremendous pain as the only good way to do it right now is by hand which is faulty and error prone," while P67 wanted, "Automated generation of simulation environments under some [custom] defined standards," because "The automated simulation environment generation is not easy. Plenty of handy work must be done by human operators."

Resource costs Simulation is computationally intensive. It often benefits from specialized hardware, such as GPUs. Participants report that hardware requirements contribute strongly to the expense of simulation. These costs are compounded when tests are run many times, such as in test automation. For example, P42 reported that difficulties including simulation in test automation include: "High hardware requirements (especially GPU-accelerated simulators) driving high cloud server costs." Participants reported problems with running simulations in parallel or using distributed computing across several machines. Participants also reported challenges in simulating large environments and simulations of long duration, as they became too resource-demanding to be practical. P67 requested, "High computational performance when the environment size grows large (Gazebo performance drops down rapidly when the number of models raises)." Participants also had issues with the cost of licenses for appropriate simulators. P66 reported that cost drove the choice not to use a given simulator: "Back then, Webots was not free," and P1 complained: "Not to mention the licensing price for small companies."

Our key insight is that 84% of participants used simulation for testing, but a lack of reproducibility, the complexities of scenario and environment construction, and considerable resource costs limit the extent of such testing.

[13]Unified Robot Description Format (URDF) is an XML file format used in robotics platforms to describe all elements of a robot.

RQ4: What challenges do developers face when using simulation for *test automation*? Research has shown that test automation can provide many benefits, including cost savings and higher software quality [105]. Despite the benefits of test automation, 27 of 69 (39.13%) participants who have used simulation for testing, reported never attempting to use simulation for automated testing. Responses indicated that the challenges with using simulation, both in general and for testing, prevented participants from attempting to incorporate it into test automation. Their reasons fell into three general categories:

1. Lack of value, where they did not find test automation valuable or necessary. As P24 mentioned "There were no obvious test harnesses available for the simulation environments I use and it did not seem obviously valuable enough to implement myself."
2. Distrust of simulation, in which the limitations of simulation itself drove the decision to not use it in automated testing. *Reality gap* and *lacking capabilities*, discussed earlier, contribute to this belief. P33 mentioned, "[Simulation is] not realistic enough for accurately modeling task; preferred running on real robot," and P20 believed that "Without a good software engineering skills the simulated environment will not replicate the real environment."
3. Time and resource limitations, where the complexity of the simulator and resource costs prevented them from attempting test automation. P18 explained "[We did not attempt to use software-based simulation for automated testing] due to the amount of time needed to setup the automated testing. Even if I think on the long term it cuts down development time, my company does not allocate time for that." and P17 simply reported that "[it] seemed very hard to do."

Among 42 people who attempted to use simulation as part of their test automation, 33 (78.57%) reported difficulties. Based on their descriptions of these difficulties, we identified the following four challenges specifically affecting test automation:

Automation features Although a GUI is an important component of a simulator, participants reported a preference towards running the simulator headless (i.e., without the GUI) when used for test automation. Disabling the GUI should reduce the computational overhead of the simulator by avoiding the need to render computation-heavy graphical models. Not being able to run a simulator headless is one of the major difficulties our participants faced in automation.

> "Making the simulator run without GUI on our Jenkins server turned out to be more difficult than expected. We ended up having to connect a physical display to the server machine in order to run the simulation properly." – P37

Furthermore, the ability to set up, monitor, and interact with a simulation via scripting, without the need for manual intervention, is vital for automation. Our participants reported devising creative solutions in the absence of scripting support. P8 shared, "Ursim[14] needs click-automation to run without human interaction." In other words, they needed to use a tool that simulated mouse clicks to run the simulator automatically.

[14]Universal Robots Simulator `https://www.universal-robots.com`

Continuous integration (CI) CI is emerging as one of the most successful techniques in automated software maintenance. CI systems can automate the building, testing, and deployment of software. Research has shown that CI practices have a positive effect on software quality and productivity [129].

CI, by definition, is an automated method, and in many cases involves the use of cloud services such as TravisCI. Our participants faced difficulties engineering the simulation to be used in CI and run on cloud servers. For example, P66 shared "I wasn't able to setup a CI pipeline which runs in GPU machines, for use with rendering sensors."

Many of these difficulties arise from missing automation features (e.g., headless execution) and high resource costs (e.g., requiring expensive, GPU-heavy hardware), discussed earlier. P77 reported "It is expensive to spin up cloud GPU VMs to run the simulator."

Simulator reliability One of the challenges of using a simulator in a test automation pipeline is the reliability of the simulator itself. In other words, participants reported facing unexpected crashes, and timing and synchronization issues while using the simulator in automation. Robotics systems, as timing-sensitive CPSs, can have their behavior distorted when timing is distorted, such as when messages arrive out-of-order or are dropped or deadlines are missed. P29, P54, P73, and P80 all reported software stability and timing issues as difficulties they faced for automation. P29 further elaborated difficulty in ensuring a clean termination of the simulator. That is, when the simulator crashes, it should properly store the logs and results before termination of the simulation, and properly kill all processes to prevent resource leaks. Clean termination is particularly relevant to test automation. Resource leaks can compound when simulations are repeated, as they are in automated testing. Compounded resource leaks can reach the point where they interfere with the ability to run additional simulations and require manual intervention.

Interface stability The stability of the simulator interface can significantly impact the automation process because inconsistent simulator API can lead to failures in client applications [280]. Participants reported unstable and fragile interfaces as a challenge for automation. For example, P39 mentioned "APIs are pretty fragile and a lot of engineering need to be done to get it working."

Five participants reported difficulties in integrating existing code or infrastructure with simulation API. P80 mentioned changing the entire physics engine source code for an application. Participants specifically desired better integration with the ROS. For example, P74 shared "I would like that [the simulator] can be used with ROS."

Our key insight is that developers want to include simulation as part of their test automation, but most developers who attempt to do so face numerous difficulties. These difficulties include an absence of automation features and a lack of reliability and API stability. Ultimately, these challenges discourage developers from using simulation for test automation, limit the extent to which it is used, and prevent developers from leveraging the benefits of CI.

3.3.3 Discussion

As robots and their associated codebases become larger and more complex, the need for, and cost of, continuous verification and validation will increase considerably. While field testing is popular it will be unable to handle the increased needs by itself because it is limited in practice by expense, hardware, human resources, and safety [20]. Simulation-based testing may serve as a cheaper, safer, and more reliable alternative by offering a means of scalable test automation [105]. Indeed, 61% of our survey participants reported that they had attempted to use simulation as part of test automation, indicating that practitioners have a strong interest in simulation-based testing.

Despite widespread interest, a multitude of challenges prevent developers from readily enjoying the benefits of simulation-based testing. A number of these challenges, such as the need for sufficient physical fidelity (i.e., the reality gap), are *inherent challenges* of building a simulator. Such challenges are well studied and widely recognized by the robotics and simulation community, and therefore outside the purview of our recommendations. While the *reality gap* can never fully be solved, the robotics and simulation community can study the trade offs inherent in usefully modeling the aspects that are important to simulating relevant systems.

However, there are ***incidental challenges of testability***, which are understudied but just as important as the inherent challenges and also impede effective simulation-based testing. These challenges include, but are not limited to, the ability to reliably run simulations without the expensive and unnecessary overhead of visualization (i.e., headless execution); the need for a powerful, expressive language for constructing realistic environments and scenarios; the ability to set up, monitor, and interact with the simulation via scripting; and stability in the simulator's client interface. Interestingly, when participants were asked to choose the simulator features that they found most useful for testing, features related to testability (e.g., "custom plugins", "exposed APIs", and "recording and logging") appeared as more popular than features related to the underlying simulation (e.g., "advanced graphics" and "high-performance physics engines").[15] Addressing these incidental challenges can enable robotics developers to better take advantage of automated simulation-based testing.

Addressing the incidental challenges requires varying levels of effort and expertise. Certain challenges can be largely fixed with engineering effort and application of best practices from the software engineering community. For example, *automation features* (e.g., headless operation), *continuous integration* support, and *simulator reliability* are all mainly engineering challenges. *Scenario construction* can be addressed through the development and application of domain-specific languages.

In other cases, the challenges can be addressed but are subject to certain limitations. For example, while it may not be realistic to expect indefinite *interface stability* from software that must change to fit evolving needs, some problems may be ameliorated by following good API design and documentation best practices [52] and engineering for backwards compatibility. Challenges in this category also include *lacking capabilities*, *reproducibility*, *resource costs*, and *complexity*. For *lacking capabilities*, it is unlikely that one simulator would provide all capabilities needed for every possible use case, but it is more realistic to engineer simulators that are extensible or tailored for the capabilities needed for individual use cases. While it would be possible to control

[15]The full list of these features and their ranking according to the participants can be found as part of our supplementary material.

some factors to create *reproducibility*, there is always a trade off against realistically modeling relevant nondeterministic phenomena. For *resource costs*, there is a trade off between the desired property of high fidelity and the corresponding resource cost. Because of the inherent complexity of the tasks needed in simulation, there will always be a degree of *complexity* in learning the simulator. The complexity and learning curve can be lessened with good design and documentation but will never disappear entirely.

From participant responses and our own experience, we observe that most popular simulation platforms (e.g., Gazebo) are predominantly designed to support manual testing during the earlier stages of robot development; test automation, on the other hand, does not appear to have been considered as an explicit use case by popular simulation platforms. To support scalable automation of simulation-based testing as part of continuous process of verification and validation, simulators should address the incidental challenges discussed earlier. We believe that the software engineering and testing communities are well-equipped to address these incidental challenges as they have studied similar problems in other domains such as distributed systems [53, 79, 170, 180]. We call upon these communities to work alongside robotics and simulator developers to study and address these incidental testability challenges.

3.4 Summary and Future Work

The focus of this thesis is on improving the state of testing and quality assurance that is performed on robotic and CPSs. As these systems have special features, we need to first develop an understanding of the challenges of testing them, and second, identify the possible avenues for performing more scalable, automated testing on them.

To summarize, in this chapter, I presented three empirical studies that investigate the state of testing and automated testing for robotic and CPSs. In Section 3.1, I performed a set of semi-structured interviews with robotic practitioners to better understand the state of (automated) testing in robotics, and I identified 9 main challenges that are barriers to testing these systems. One of the important takeaways of this work is that 1) field testing is the predominant means of quality assurance for these systems, and 2) robotics practitioners generally distrust simulation even though they are aware of the theoretical benefits of using simulation for automated testing.

In Section 3.2, I investigate the validity of concerns about simulation, and the potential impact of using low-fidelity software-based simulation on exposing failures in robotics systems by conducting a case study. In this study, I showed that low-fidelity simulation can be an effective approach in detecting bugs and errors with low cost in robotic systems.

Given the findings of these two studies, the question remains: why simulation-based testing is not widely adopted as a mean of automated testing of robotic systems? In Section 3.3, I presented a large-scale survey with robotics practitioners to identify features in robotics simulators that are the most important for automated testing, and the challenges of using these simulators. In total, I presented 10 challenges that limit the extent to which simulators are used for different purposes including automated testing. The main takeaway of this study is that robotic simulators are generally not well-equipped to be used in test automation.

Even though these studies explore the state of testing and automated, simulation-based testing of robotic and CPSs to some extent, there is room for more studies to be conducted in this

area in the future, to pinpoint the barriers of automated testing in this field, and propose solutions to address these challenges. For instance, future studies can explore the state of (automated) hardware-in-the-loop (HITL) simulation-based testing, and identify its advantages and disadvantages over using SITL simulation-based testing.

In addition, future studies can focus on providing a set of guidelines and starting points on deploying automated testing or continuous integration on robotic systems for robotic practitioners, who do not necessarily have the experience and background in this area. Alami et al. [26] have started developing such guidelines for ROS systems, and a set of best practices has been introduced on ROS Wiki page. [16] However, these guidelines are not widely used and can be improved in the future. We can even imagine tools that use automated or semi-automated static and dynamic analysis techniques to provide assistance to robotics developers, who fancy to deploy automated testing on their systems.

Finally, in this chapter I discussed many challenges that involve testing of robotic and CPSs. Future studies can focus on addressing these challenges, and as mentioned earlier, software engineering community is well-equipped to resolve or make improvements to many of these challenges. For example, improving robotic simulators by adding automation features and adjustable controls over noise and non-determinism in simulations.

[16] `http://wiki.ros.org/BestPractices`

Chapter 4

Automated Oracle Inference

As presented in Figure 1.1, an oracle that can automatically distinguish between correct and incorrect behaviors of the system is essential to achieve an automated testing pipeline. However, generating an oracle is a well-known problem in software engineering [41], and as presented in Section 3.1, is one of the challenges of testing robotics systems. Although many approaches have been proposed to address the oracle problem for pure software (cyber) systems [19, 149, 182, 188, 225, 296], they are not appropriate for robotics and cyberphysical systems (CPSs) for the following reasons:

1. CPSs often contain proprietary third-party components (such as cameras or other sensors) for which source code is unavailable, and so techniques should minimize or avoid relying on source code access.
2. CPSs are inherently non-deterministic due to noise in both their physical (e.g., sensors, actuators, feedback loops) and cyber components (e.g., timing, thread interleaving, random algorithms) and may react to a given command in a potentially infinite number of subtly different ways that are considered to be acceptable. That is, for a given input and operating environment, there is no single, discrete response that is correct, but rather an envelope of responses that are deemed correct. And so techniques should be robust to small, inherent deviations in behavior.
3. The CPS may respond differently to a given instruction based on its environment, configuration, and other factors (i.e., its operating context). For example, a flying copter may refuse to fly to the specified point if its battery is depleted. And so, techniques must be capable of capturing contextual behaviors for a given command.

To tackle the above mentioned challenges, I propose Mithra: a novel oracle learning approach, which identifies patterns of normal (common) behaviors of the system by applying a multi-step clustering approach to the telemetry logs collected over many executions of the system in simulation. Mithra uses the identified clusters as the core of its oracle and determines correctness of system's executions based on their similarity to identified clusters. It tackles all three above mentioned challenges as it does not require source code access, avoids over generalization and is robust to small deviations from expected behavior, and identifies contextual behaviors.

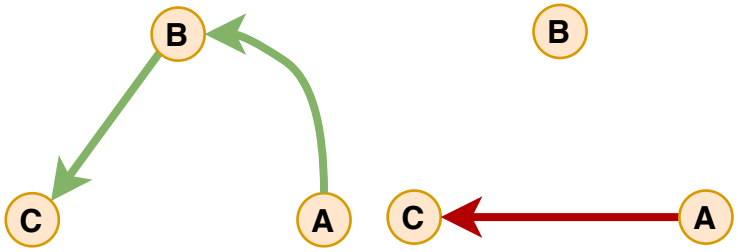


Figure 4.1: A simplified depiction of the motivating example (ARDUPILOT's Issue #9657). The left figure illustrates the intended path of the vehicle from A to C via B. The vehicle is instructed to travel along a spline between A and B, before continuing along a straight line between B and C. The right figure illustrates the actual, erroneous path of the vehicle. The issue causes the vehicle to skip the spline waypoint B, and travel directly from A to C along a straight line.

4.1 Case Study

As a running example, I use ARDUPILOT system described is Section 2.2.

4.1.1 Motivating Scenario

As described in Section 2.2, ARDUPILOT is a mature autopilot software for CPSs that is used in a wide variety of vehicles and environments that are either in, or approaching, deployment. Although ARDUPILOT is functionally stable and used by over one million vehicles [1], it continues to evolve, and new issues and erroneous behaviors are continually discovered and reported over time. In 2019 alone, 722 new issues were filed on ARDUPILOT's issue tracker, of which 130 were labeled as bugs. Many of these bugs, such as the one described in Issue #9657, occur only under specific conditions and may result in behavioral changes that may have not been considered by ArduPilot's testing team.[1] In the case of Issue #9657, the vehicle misbehaves when instructed to navigate a series of waypoints that includes a spline path. By default, the vehicle will travel along a straight line between waypoints. However, operators may also instruct the vehicle to traverse a smooth path between waypoints along a spline. In the relatively rare event that a series of waypoints includes a spline path, the vehicle will erroneously skip the first waypoint along a spline path (Figure 4.1).

Identifying such bugs requires both a means of **triggering** the bug (i.e., subjecting the system to a particular scenario and environment), and **detecting** that a failure has occurred (i.e., the system behaves in an unintended manner). Numerous studies on automated test input generation have focused on addressing the **triggering** problem [110, 122, 197, 205, 268, 269, 277]. Using artifacts and models of the system, or a search-based approach, these studies propose ideas on generating test inputs, scenarios, and environments that trigger and expose different behaviors of the system. In this work, we assume a means of triggering bugs and focus our attention on the problem of automatically **detecting** failures.

The example of ARDUPILOT and Issue #9657 motivates our approach in creating oracles

[1]Issue: `https://github.com/ArduPilot/ardupilot/issues/9657` fixed by pull request `https://github.com/ArduPilot/ardupilot/pull/10338` [Date Accessed: September 2, 2020]

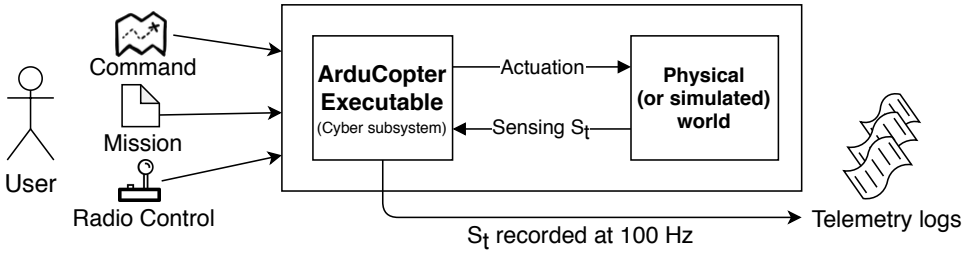


Figure 4.2: A simplified view of the ArduCopter communications architecture. Input is provided by the user to the cyber component in the form of discrete commands and missions, or as a continuous radio control signal. The cyber component sends signals to actuate the physical component of the system, and reads sensor values. The state of the system, reported by the sensors, is periodically written to a telemetry log.

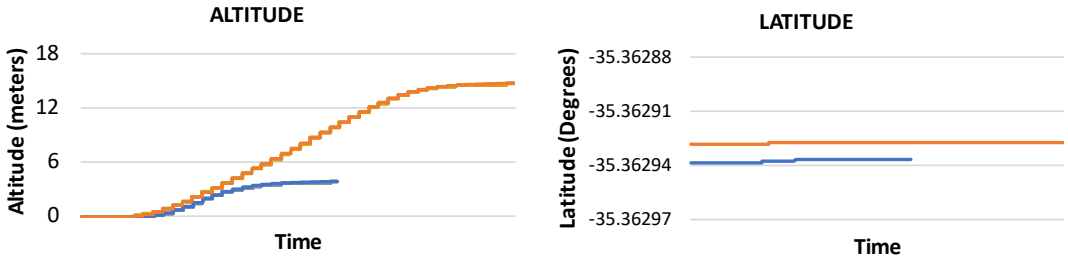


Figure 4.3: An example of two execution traces for the ARDUCOPTER's TAKEOFF command with respect to its ALTITUDE and LATITUDE state variables. In the bottom trace (blue), TAKEOFF(ALT:4.0), the copter elevates 4 meters above the ground. In the top trace (orange), TAKEOFF(ALT:14.7), the copter elevates 14.7 meters above the ground. In both cases, the LATITUDE remains roughly fixed.

for mature systems. As mentioned, a mature software (e.g., ARDUPILOT) performs common scenarios as expected. For example, when a vehicle is instructed to navigate to a target location, it performs as expected under most conditions. Note that if such common behavior becomes faulty in a mature system, the maintainers and testers would be alerted quickly, as it affects many users and scenarios. However, in circumstances involving behaviors that are less commonly used, such as scenarios that involve spline waypoints, the vehicle may misbehave and perform not exactly as expected.

In this work, we use a novel clustering approach to automatically identify the common behaviors of the system, which we use to form an oracle that can distinguish between expected and unexpected executions.

4.1.2 ARDUCOPTER's Architecture

For our running example, we use ARDUPILOT (version COPTER-3.6.9) as the controller for a simulated quadcopter. Figure 4.2 provides a simplified view of the cyber and physical components of ARDUCOPTER. The user provides input to ARDUCOPTER's cyber component in one of three forms: (1) as a discrete *command* from a ground control station, such as TAKEOFF,

along with a set of parameters (e.g., desired altitude); (2) as a precomputed sequence of such commands, known as a *mission*; or (3) in the form of a continuous sequence of radio control signals. The cyber component of ARDUCOPTER interacts with the physical component by polling its sensors at a fixed interval (e.g., once every 10ms) to determine its *extrinsic state* and sending signals to its actuators based on its extrinsic state and the user-provided input. The extrinsic state $\boldsymbol{s}_t$ of the system at time t describes the values of its *state variables*, each representing the value of a particular sensor, and is composed of both continuous (e.g., VELOCITY) and categorical values (e.g., STATUS). The cyber component of the system periodically logs its extrinsic state to a *telemetry log* at a fixed rate (e.g. 10 Hz). From the telemetry log, we extract an *execution trace* S for each command execution that records the sequence of extrinsic states logged during execution. We use the execution trace as input to our technique.

Figure 4.3 provides a simplified example of two execution traces for the TAKEOFF command. Each execution trace can be represented as a *heterogeneous multivariate time series*: time series data consisting of multiple dimensions that include both continuous and nominal data. Since the time taken to complete an execution may vary, traces are variable in length and may consist of thousands of recorded states. For example, a 30-second execution of a single command results in a trace with 300 state observations if telemetry is recorded at 10 Hz. On another execution, the same command may take 50 seconds to complete and result in 500 state observations.

In this work, we restrict our attention to command-based user inputs and leave an application of our approach on continuous inputs to future work. We consider 10 out of 25 commands supported by the ARDUCOPTER mission planner,[2] and 18 associated state variables describing properties like orientation, position, and velocity. The 15 excluded commands are specific to particular hardware (e.g., DO-DIGICAM-CONTROL triggers the camera shutter if the copter is mounted with a camera).

4.2 Clustering Multivariate Time Series

Statistical machine learning approaches deal with the problem of finding a predictive function based on data. These approaches are applied on a collection of instances of the data, which acts as the input to the learning process (i.e., training data). What the algorithm can learn from the training data varies in different approaches [295].

Supervised learning methods take a collection of training data with given labels (e.g., "male" and "female"), and learn a predictive model $y = f(x)$, which can predict the label (y) of a given input (x). Depending on the domain of label y, supervised learning problems are further divided into *classification* and *regression*. Classification is the supervised learning problem with discrete classes of labels, while regression is applied on continuous labels. Support vector machines (SVM), decision trees, linear and logistic regressions, and neural networks are all examples of supervised learning algorithms [163].

Unsupervised learning techniques work on an *unlabeled* training data. Common unsupervised learning tasks include: (1) clustering, where the goal is to separate the n instances into groups, (2) novelty detection, which identifies the few instances that are very different from the

[2] `http://ardupilot.org/copter/docs/mission-command-list.html`
[Date Accessed: September 2, 2020]

majority, and (3) dimensionality reduction, which aims to represent each instance with a lower dimensional feature vector while maintaining key characteristics of the training sample. Clustering approaches such as k-means [181], k-medoids [152] and hierarchical clustering [143], in general split the training data into k clusters, such that instances in the same cluster are similar, according to a similarity measure, and instances in different clusters are dissimilar. The number of clusters k may be specified by the user, or may be inferred from the training sample itself [283].

Our oracle learning approach builds oracles by clustering telemetry logs represented by multivariate time series (MTS). In this section, we provide the necessary background in MTS clustering to understand the techniques underlying our approach. Time series clustering has widely been used to find common patterns in streams of data in a variety of domains including bioinformatics and biology, genetics, finance, air quality control, and meterology [37, 64, 69, 253]; this work presents a novel formulation that effectively models correct CPS behavior.

k-Medoids The k-medoids algorithm [152] is a clustering technique that uses a given distance metric to partition a given dataset into k clusters such that the distance between the points within a cluster and the center of that cluster (i.e., the *centroid*) is minimized. Unlike the well-known k-means, in which the center of a cluster is the average between its points, k-medoids uses an existing representative point within the cluster as its center. k-medoids is attractive for clustering MTS datasets because it does not introduce additional, expensive distance calculations (e.g., measuring the distance between a given point and the mean of a cluster, as in k-means). k-medoids only compares existing points to one another, and so a distance matrix can be efficiently precomputed.

Distance Metrics Any clustering approach requires a suitable distance metric. A common distance metric is Euclidean distance (i.e., L2 norm), which is inexpensive to compute. However, Euclidean distance can only be used for *same-length* MTS (i.e., time series of an equal duration and number of observations). In our case, where this assumption does not hold, we require an alternative distance metric. We discuss two alternaive metrics that can compare variable-length MTS: Dynamic Time Warping [45] and Eros [286].

Dynamic Time Warping (DTW) [45, 116, 147, 148] is a similarity measure[3] that compares temporal sequences (i.e., traces) in terms of their "shape". DTW accounts for variations in duration, length, speed, and amplitude between two traces by mapping points from one trace to another trace via a non-linear process of "warping", illustrated in Figure 4.4. DTW computes the optimal mapping between A and B such that every point in A is mapped to at least one point in B and vice versa in such a way that the order of points is retained, and the sum of distances between mapped points is minimized.

Although DTW provides a powerful means of comparing variable-length k-dimensional time series, it comes at the cost of a considerably $O(kmn)$ higher runtime complexity compared to $O(kn)$ complexity of the L2 norm, where m and n are the lengths of two time series.

[3]Note that although DTW measures a distance-like quantity, it is not a *true* distance metric since it violates the triangle inequality: $d(\boldsymbol{x}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z})$.

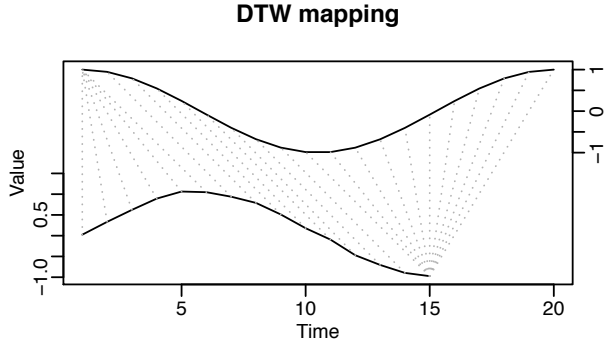


Figure 4.4: Dynamic time warping measures the distance between two time series of unequal length by mapping the points in each time series onto the other via warping. The solid lines represent individual time series and the dashed lines represent the warping that maps one onto the other. By warping, DTW allows similarity between time series to be computed based on their *shape.*

This can be reduced using a DTW approximation or lower bound such as FastDTW [245] or LB_Keogh [153].

The Extended Frobenius norm [286], or Eros, is a cheaper alternative to DTW that uses Principal Component Analysis (PCA) [17, 132, 226] to measure the distance between two variable-length MTS. Instead of measuring similarity between them by aggregating similarities between their individual variables, Eros treats each MTS as a matrix and uses the principal components to measure similarity.

Given an MTS dataset, Eros first determines the eigenvectors and eigenvalues of the covariance matrices of each MTS within the dataset. Eros then aggregates the eigenvalues to obtain *weights* for the dataset. Finally, Eros uses those weights to measure the similarity between two MTS in terms of their associated eigenvectors.

Eros is considerably cheaper to compute than DTW with an amortized runtime complexity that is linear in the number of variables in the MTS, and unlike Euclidean distance, can be applied to variable-length MTS. Eros can account for differences in shape and is capable of handling shifts in time, but unlike DTW, it does not account for scaling over time.

4.3 Approach

In this section, I describe Mithra, our proposed unsupervised oracle learning approach, based on anomaly detection for mature cyberphysical systems. Mithra learns oracles for CPSs that accept a vocabulary of discrete commands, and produce telemetry logs (e.g., ARDUCOPTER). It does this by using a training set of telemetry logs to identify clusters representing the qualitatively different behaviors for each command. For example, based on traces such as those in Figure 4.3, Mithra detects one such common behavior, TAKEOFF(ALT:<p_{alt}>), in which ALTITUDE gradually increases until reaching a specified altitude p_{alt} while LATITUDE remains constant.

Approaches for clustering and classifying time series that are based on comparing differences in shape are often superior in terms of performance than those that compare differences in time [24, 231]. Unfortunately, clustering strictly with a DTW distance measure does not scale to

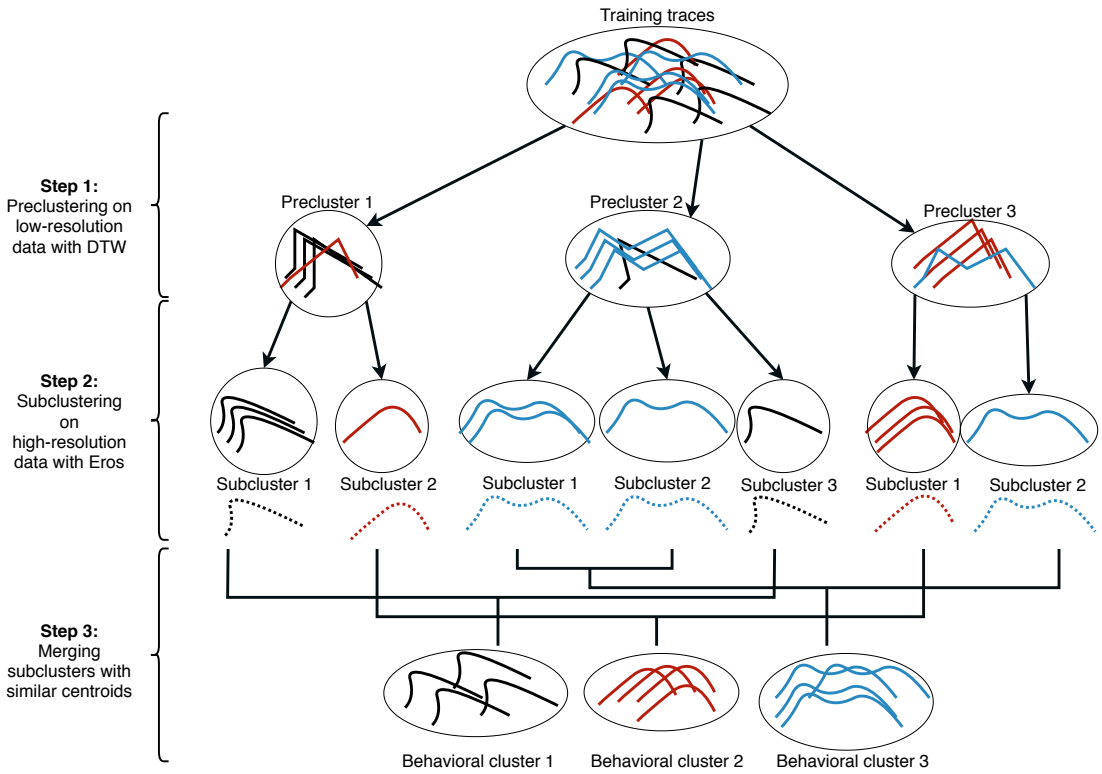


Figure 4.5: An overview of Mithra's three-step clustering approach of preclustering, subclustering, and merging. Solid lines represent individual traces, and dashed lines represent cluster centroids.

large datasets. As a result, Mithra clusters execution traces based on overall shape using a three-step approach inspired by Aghabozorgi et al.'s method for clustering large time-series data [23]. Figure 4.5 provides a high-level overview:

1. *Preclustering*: A low-resolution version of the training data is clustered into *preclusters* to reduce the search space.
2. *Purifying*: As the low-resolution preclusters are insufficiently accurate, Mithra next creates a set of *subclusters* for each precluster using high-resolution data.
3. *Merging*: Similar subclusters are merged to obtain a set of *behavioral clusters*, producing a simpler model that is cheap to query.

Using its learned *behavioral clusters*, Mithra constructs an oracle for each command based on anomaly detection, that marks execution traces as either CORRECT or ERRONEOUS based upon their similarity to the contextual behaviors represented by those clusters.

Note that although the structure of our technique draws inspiration from the prior work, Aghabozorgi et al.'s approach [23] can only be applied to datasets of time series with fixed length, and thus is not suitable off-the-shelf for our problem domain. We therefore propose key novelties at each clustering step to enable scalable clustering of variable-length data.

4.3.1 Training Data

In the training phase, Mithra takes, as input, a set of telemetry logs. Ideally, the set should contain logs that exercise all functionality of the system, covering a diversity of possible scenarios, though this is not a strict requirement. Mithra constructs an individual training set for each command within the vocabulary of the CPS by extracting the relevant execution traces for that command from the provided set of telemetry logs.

Note that, like most other techniques [14, 30, 124, 136, 210], Mithra is unsupervised. Thus, these training logs are not labeled in terms of whether they correspond to correct or erroneous behaviors. Furthermore, Mithra is robust to erroneous traces within the training set provided that they are rare. This is also consistent with the prior techniques, which make the same assumption; fortunately, most programs behave correctly most of the time, and erroneous behavior is indeed typically rare [85].

Mithra preprocesses training data in three ways:

1. **Converting Categorical Data.** Categorical variables (e.g., ARDUCOPTER's MODE, which takes values such as `STABILIZE`, `AUTO`, and `GUIDED`), complicate distance measures, as the distance between two categorical datapoints can only be measured by whether they take the same value. Mithra converts categorical data to numerical data using one-hot encoding [70], where each category is turned into a dimension with binary value.
2. **Normalization.** Since it may not be meaningful to compare different state variables (e.g., VELOCITY and LATITUDE) due to differing ranges and units, we standardize [117] the data to ensure that differences in each state variable are treated with equal importance. Each dimension (i.e., state variable) within a time series is scaled to resemble a normal distribution with $\mu = 0$ and $\sigma = 1$.
3. **Feature Selection.** Clustering techniques can suffer from the *curse of dimensionality* on datasets with many dimensions [62]. Therefore, Mithra accepts an option to select N_{FEATURES} dimensions in the training data with the highest entropy as a preprocessing step. High entropy in a dimension indicates that it can be informative in distinguishing different behaviors. We leave investigation of other feature selection approaches to future work.

4.3.2 Oracle Learning

Given a training set of traces for a command, Mithra attempts to identify the set of contextual (i.e., disjunctive) behaviors for that command. Mithra uses a three-step time series clustering approach that allows clustering to scale to a large number of detailed traces:

Step 1: Preclustering Mithra first downsamples the training execution traces to produce a set of low-resolution traces to be clustered. By reducing the resolution of the data, Mithra can more efficiently compute DTW distance on an approximation of its input traces. The goal of this step is to reduce the search space for the subsequent, more computationally intensive steps.

To lower trace resolution, Mithra uniformly drops data points from each time series. For example, trace $t = [S_0, S_1, ..., S_{100}]$ with 101 data points can be downsampled to a lower-resolution trace $t' = [S_0, S_5, S_{10}, ..., S_{95}, S_{100}]$ with 21 data points. Even though t' does not represent the

exact behavior of trace t, it approximates t's shape and can be used to create an initial set of *preclusters*.

To obtain the set of preclusters, Mithra applies k-medoids clustering to the low-resolution data using FastDTW [245] as its distance metric. The number of clusters k is obtained dynamically by finding the $1 < k < k_{max}$ that maximizes the silhouette score [239].

Step 2: Purifying Since low-resolution data is used to obtain the set of preclusters, those preclusters may represent spurious patterns that do not hold on the original, high-resolution data. Therefore, in the second step, Mithra divides the contents of each precluster into multiple subclusters based on their Eros similarity. Although Eros is a less effective means of measuring similarity between traces than DTW (i.e., scale information is lost), it is inexpensive to compute and provides useful partial information about similarities in shape. To calculate the subclusters, we apply k-medoids clustering within each precluster, and find the *medoid* that is most representative of all traces within a subcluster.

Step 3: Merging Finally, Mithra uses FastDTW to merge subclusters that share a similar shape based on the original, high-resolution data. This step prevents representation of the same contextual behavior by multiple subclusters, which leads to a simpler model that is cheaper to query. To do so, Mithra first computes the DTW distance among the medoids of subclusters using the original, high-resolution traces for those medoids. Although the time series are more detailed than those used during preclustering, the total number of time series, and, by extension, distance calculations, is far smaller, ensuring this step is scalable.

Mithra then uses the computed medoid distances to reduce the set of subclusters into a set of *behavioral clusters* by merging subclusters that share highly similar medoids. Mithra uses hierarchical clustering [143] to find the sets of similar subclusters. For every set of similar subclusters, Mithra constructs a new behavioral cluster that includes all their traces, and applies DTW averaging with uniform scaling [98] to the medoids of those subclusters to produce a centroid that best represents all traces in the new behavioral cluster.

Finally, Mithra uses FastDTW to compute μ_β and σ_β for each behavioral cluster β based on the distance from the traces within β to the centroid of that cluster $\boldsymbol{c}_\beta$, which Mithra uses to construct the decision boundary for β.

4.3.3 Oracle Querying

The behavioral clusters for each command represent qualitatively different modes of behavior observed for that command. These may include both behaviors that are frequently observed and assumed to be correct (e.g., clusters with more than one hundred traces), as well as behaviors that are rarely observed and suspected to be erroneous (e.g., clusters with fewer than five traces).

Mithra uses the behavioral clusters to predict whether a *new* trace is CORRECT or ERRONEOUS by comparing it to the centroid of its best-fit cluster. More formally, given a previously unseen execution trace τ for a command, Mithra first finds the behavioral cluster $\beta^*_\tau \in \mathrm{BC}$ that most closely resembles τ based on the DTW distance between τ and the centroid of each cluster:

$$\beta^*_\tau = \arg\min_{\beta \in \text{BC}} DTW(\tau, \boldsymbol{c}_\beta)$$

Mithra then uses β^*_τ to predict the label ℓ_τ for that trace as:

$$\ell_\tau = \begin{cases} \text{ERRONEOUS} & \text{if } |\beta^*_\tau| < \rho \\ \text{ERRONEOUS} & \text{if } DTW(\tau, \boldsymbol{c}_{\beta^*_\tau}) > \mu_{\beta^*_\tau} + \theta\sigma_{\beta^*_\tau} \\ \text{CORRECT} & \text{otherwise} \end{cases}$$

where $|\beta|$ is the number of traces within β, $\rho \in \mathbb{Z}^+$ is the *rarity threshold*, and $\theta \in \mathbb{R}^+$ is the *acceptance rate*. If β^*_τ contains fewer than ρ traces, it is assumed to represent a rare, and thus, erroneous behavior, and so, τ is marked as ERRONEOUS. The rarity threshold allows Mithra to be more robust towards erroneous traces in the training data. In the more likely case where β^*_τ contains at least ρ traces, then β^*_τ itself is assumed to represent a common, and thus, correct behavior. In that case, Mithra uses the precomputed DTW distance to determine whether τ lies within the decision boundary of β^*_τ, and if so, labels it as CORRECT. The acceptance rate θ is used to alter the extent of the decision boundary and provides the user with a means of controlling the precision-recall tradeoff of the classifier to their preferences. I investigate and discuss the effects of θ in Section 4.4.3.

4.3.4 Implementation

Our implementation of Mithra, which we released as part of our replication package,[4] allows tuning, such as via the level of resolution decrease for *Preclustering* step, or the N_{FEATURES} (Section 4.3.1).

One additional optional argument that can improve Mithra's performance is parameter handling. The behavior of a CPS with respect to a certain command often depends upon the parameters provided to that command. In the example of Figure 4.3, if the copter flies to altitude of 10 meters instead of 4 when instructed to TAKEOFF(ALT:4.0), the trace should be marked as ERRONEOUS. However, by default, Mithra cannot connect two relevant dimensions in the traces (in this case, the ALTITUDE of the copter and the parameter passed to the command p_{alt}).

To account for parameter values, we can add new dimensions to input traces that are dynamically computed using the values of parameters and state variables. For example, for the command TAKEOFF(ALT:<p_{alt}>), we dynamically compute a variable DIST_ALT as $(p_{alt} - \text{ALTITUDE})$. With this new dimension, Mithra's learned clusters represent that, for example, in CORRECT TAKEOFF(<p_{alt}>) traces, the value of DIST_ALT always converges to zero; we can mark ERRONEOUS cases where it does not (e.g., flying to 2 meters altitude when 5 is given as the parameter). Note that this added dimension does not specify the correct or expected behavior; it merely expresses a meaningful connection between parameters and state variables.

The definitions for dynamically computed dimensions are presently user-provided. As the number of command parameters is usually very limited and many commands share the same set of parameters, specifying these definitions is fairly simple. For our case study of ArduPilot,

[4] `https://bit.ly/2S9m7cd`

we specify definitions for 4 dynamically computed dimensions that are shared among 7 of 10 commands. The definitions for these added dimensions are provided as part of our replication package. Note that Mithra *can* operate without these additional dimensions, but it will be less accurate. We anticipate that such dimensions are likely automatically discoverable, a prospect that we leave to future work.

4.4 Evaluation

To determine whether our technique is an effective oracle learning method for mature cyber-physical systems, we conducted experiments, outlined in Section 4.4.2, on the case study system (i.e. ARDUCOPTER). We compare Mithra to the state-of-the-art [124] (AR-SI, described in Section 4.4.1). We answered the following research questions:

RQ1 (Accuracy) How accurately does our clustering method distinguish between correct and erroneous traces?

RQ2 (Comparison) How does the labeling accuracy of Mithra compare to AR-SI [124], a state-of-the-art oracle learning approach for cyberphysical systems?

RQ3 (Conceptual Validation) How do Mithra's individual steps influence its overall labeling accuracy?

RQ4 (Time) How long does it take to train and query Mithra, and how does it compare to AR-SI?

Finally, we evaluated Mithra on an autonomous racing CPS in Section 4.4.7 to show its applicability to systems beyond ARDUPILOT.

4.4.1 Baseline

To compare our approach with the state-of-the-art, we reimplemented He et al.'s approach for creating autoregressive system identification (AR-SI) oracles for CPSs [124].[5] Like our approach, AR-SI does not assume source code access and does not require training on a bug-free, ground-truth version of the CPS. Based on the assumption that many CPSs are designed to run *smoothly* when noise is under control, AR-SI determines whether a trace is erroneous or correct by checking the smoothness of the system's behavior. Let $Y_i \in \mathbb{R}^m$ represent the state of the system at time i with m state variables, and $U \in \mathbb{R}^q$ represent user input (i.e., command parameters). AR-SI models the relationship between U and Y_i as follows:

$$Y_i = (\sum_{j=1}^{p} A_j Y_{i-j}) + BU + \xi_i \tag{4.1}$$

and optimizes model parameters $A_1, A_2, ..., A_p \in \mathbb{R}^{m \times m}$ and $B \in \mathbb{R}^{m \times q}$ so that the runtime accumulated SI error energy ξ_i is minimized. Then, AR-SI uses the optimal model parameters ($A_1^*, A_2^*, ..., A_p^*$ and B^*) to predict the next state of the system Y_{i+1}:

[5]The source code of AR-SI is not publicly available, and we were unable to gain access via private email correspondence.

$$\hat{Y}_{i+1} = (\sum_{j=1}^{p} A_j^* Y_{i+1-j}) + B^* U \tag{4.2}$$

and collects the prediction error as $e_{i+1} = \hat{Y}_{i+1} - Y_{i+1}$.

In other words, AR-SI uses the past p observed states of the system to predict its next state with the assumption that state changes tend to be smooth and the prediction error should be low. When the prediction error for all states in the trace is computed, AR-SI checks whether they contain an outlier prediction error. If so, the trace is marked as ERRONEOUS, otherwise it is marked as CORRECT. Any prediction error outside of $\mu \pm 6\sigma$ is considered an outlier.

AR-SI was originally evaluated on our case study CPS; I discuss methodology next.

4.4.2 Experimental Methodology

We constructed a benchmark for our case study, ARDUPILOT, which we used to evaluate our research questions. This benchmark consists of a *training dataset* and an *evaluation dataset*. The training dataset consists of unlabeled traces for 2500 randomly generated missions; it is used to train Mithra. The evaluation dataset provides a labeled, balanced set of 233 erroneous and 233 correct traces. We used it as ground truth when measuring the accuracy of Mithra and AR-SI (i.e., the ability to discriminate between erroneous and correct traces). Note that the labels of the evaluation dataset are not provided to either approach.

To ensure reproducibility and avoid physical harm, we used software-in-the-loop (SITL) simulation to obtain traces in lieu of traces from real-world field testing. We sampled state according to the simulation clock rather than the wall clock, retaining the same information as a corresponding field trace. The practice of using simulation to obtain traces for this type of evaluation is common [29, 65, 124]. Below, I provide key details about benchmark construction.

Training Dataset As a source of training data for our technique, we recorded traces for 2500 randomly generated missions in simulation; To accelerate data collection, we spread the process across 30 cores and use 40X simulation speedup. In total, we took roughly 15 hours to collect training traces.

Evaluation Dataset We constructed our evaluation dataset by first identifying 11 historical bugs via manual investigation of issues and bug-fixing commits on the ArduPilot repository.[6] We transformed each historical bug into a controlled *bug scenario* by manually grafting the bug onto the ground-truth version of ARDUPILOT, COPTER-3.6.9. By individually grafting the bugs onto the ground-truth version, rather than using those historical versions directly, we ensure that the only differences in behavior are due to a particular bug and not from an unrelated change to the program. We generated an additional 13 bugs by applying the same historical faults to other parts of the code, raising the total number of bug scenarios to 24.

For each bug scenario, we used a hand-written mission template, tailored to that scenario, to randomly generate 10 missions that trigger and manifest the bug. After running each mission, we

[6] `https://github.com/ArduPilot/ArduPilot`

collected line coverage of the execution to ensure that the executed mission does in fact execute the lines of interest (i.e., faulty lines).

Finally, we used the generated missions to construct an evaluation set of correct and erroneous traces. We obtain 240 erroneous traces by executing each bug scenario against its associated bug-triggering missions. However, we excluded traces resulting in software crashes (e.g., segmentation faults) from our dataset since those traces can simply be labeled as ERRONEOUS and no oracle is required. We excluded 7 out of 240 traces due to system malfunction. We then obtained 233 correct traces by executing all 233 bug-triggering missions against the ground-truth version of the program, and created a balanced set of evaluation traces.

Comparison to AR-SI's Methodology AR-SI was originally evaluated on a dataset of 8 historical ArduPilot bugs and 17 artificial bugs created by fault injection [124]. Similar to our approach, He et al. collect a set of traces, which are considered ERRONEOUS if they execute the faulty lines, and CORRECT otherwise. However, the AR-SI dataset of bugs and traces is not available publicly, and we have been unable to gain access via private correspondence. Therefore, we created a dataset of 24 real-life bugs and 466 traces, and release it as a benchmark to be used by studies in the future.

To evaluate the effectiveness of AR-SI, He et al. compared AR-SI against a "human oracle" devised by CPS experts. The human oracle consists of three rules that check that the velocity and angular velocity of the copter are within certain bounds (e.g., "velocity shall never exceed $\pm$20m/s"). He et al. found that AR-SI produced fewer false positives and false negatives than the human oracle. Approximately 70% of traces that were identified as erroneous by the human oracle were, in fact, correct. We choose not to evaluate against a human oracle since its performance is dependent upon the knowledge and skills of the experts, and therefore any comparison to such an oracle would not yield meaningful insights on the performance of Mithra or AR-SI.

Setup To account for nondeterminism, we ran each experiment on 20 different seeds. For all experiments, we ran Mithra with feature selection $N_{\text{FEATURES}} = 10$, rarity threshold $\rho = 5$, and maximum number of clusters $k_{max} = 15$.

We conducted our experiments on a single machine, running Ubuntu 18.04, with the following specifications: TR 2990WX (32 cores), 64GB RAM, GTX 1080 Ti, and a 1 TB NVMe SSD.

Replication We provide our source code, raw results, scripts to analyze those results, and benchmark traces as part of our replication package: `https://bit.ly/2S9m7cd`.

Evaluation Metrics To evaluate a candidate model (i.e., the output of our technique), we iterated over each trace in the evaluation set and checked whether the label predicted by the model (i.e., CORRECT or ERRONEOUS) matched the expected label. We then computed the number of true positives TP (erroneous traces marked as ERRONEOUS), false positives FP (correct traces marked as ERRONEOUS), true negatives TN (correct traces marked as CORRECT), and false negatives FN (erroneous traces marked as CORRECT). From those values, we obtained a summary of model performance:

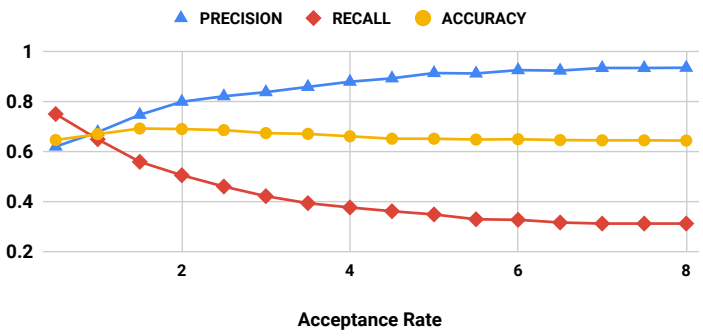


Figure 4.6: Relationship between Mithra's median precision (blue triangles), recall (red diamonds) and accuracy (yellow circles) and acceptance rate used to classify outliers.

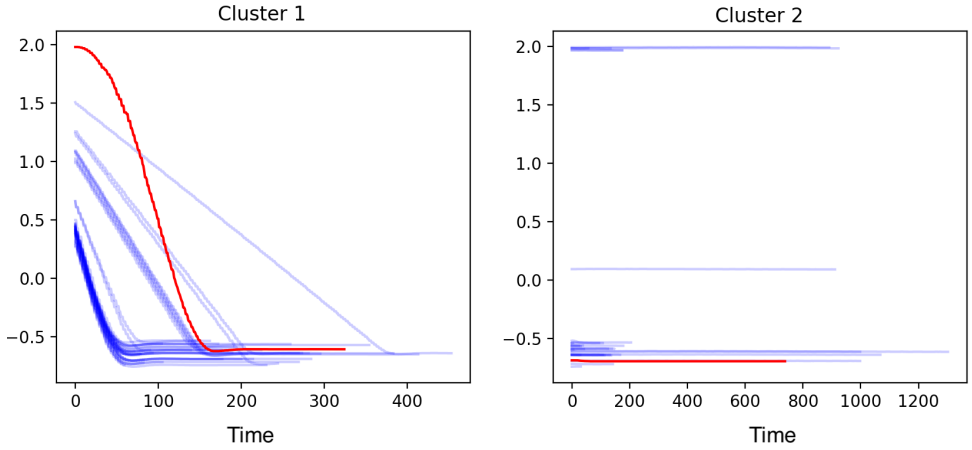


Figure 4.7: Two behavioral clusters for LOITER_TIME that were learned by Mithra, plotted with respect to normalized LATITUDE (y-axis) over time (x-axis). Each blue line represents a single trace in the cluster, and the red lines represent the centroid of the cluster. The left cluster captures the behavior of the copter moving to a specified location before loitering, whereas the right cluster shows the behavior of remaining at its current location and loitering.

Precision: fraction of traces reported as erroneous that are truly erroneous ($\frac{TP}{TP+FP}$).

Recall: fraction of erroneous traces reported as such ($\frac{TP}{TP+FN}$).

Accuracy: fraction of correctly labeled traces ($\frac{TP+TN}{TP+FP+TN+FP}$).

Note that we used accuracy rather than F1-score, defined as the harmonic mean of recall and precision, as an overall measure of performance as the F1-score places little weight on false positives and is best suited to imbalanced datasets.

4.4.3 RQ1: Accuracy

Figure 4.6 illustrates the median performance of Mithra with different acceptance rates θ. As the acceptance rate increases, recall decreases and precision increases, resulting in a more con-

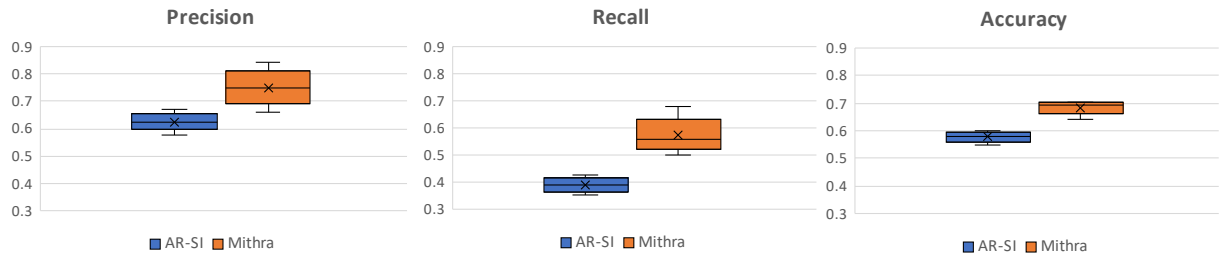


Figure 4.8: A performance comparison between AR-SI and Mithra. Using a one-sided Mann-Whitney U test, we show that Mithra outperforms AR-SI significantly ($\alpha = 0.05$) in terms of precision, recall, and accuracy.

servative model that detects fewer erroneous traces overall, but ensures that traces marked as erroneous are more likely to be truly erroneous. Overall accuracy remains fairly steady as the acceptance rate is increased, demonstrating the tradeoff between false negatives and false positives. By modifying the acceptance rate, users can customize Mithra to their preferences [142, 241].

Overall, Mithra achieves a median accuracy of 66.5% across all seeds, and reaches its highest accuracy of 69.3% when its acceptance rate $\theta = 1.5$ (marking 74.7% of truly correct traces, correct). We therefore used acceptance rate $\theta = 1.5$ for the rest of our experiments.

As an example of a correctly detected behavior for ARDUCOPTER, we took a look at the behavioral clusters for the LOITER_TIME(TIME, LAT, LON, ALT) command. According to the ARDUCOPTER documentation,[7] the behavior of LOITER_TIME is described as "The vehicle will fly to and then wait at the specified location for the specified number of seconds." However, as stated in the documentation, if the given latitude and longitude are both set to zero, the copter should hold at its current location. Figure 4.7 illustrates the behavioral clusters that were identified by Mithra for LOITER_TIME. Cluster 1 captures traces where the latitude of the copter changes drastically, whereas in Cluster 2, the latitude of the copter remains constant. In this example, we can see that Mithra automatically identifies the two correct behaviors of LOITER_TIME as stated in the documentation.

The motivating example described in Section 4.1.1 illustrates the case where the copter misbehaves on SPLINE_WAYPOINT command that is followed by another navigation command. On 20 evaluation traces (10 correct and 10 erroneous) generated for this issue, Mithra reaches median accuracy, recall and precision of 70% , 90%, and 66.6%, respectively ($\theta = 0.5$). Intuitively, this demonstrates that when Mithra is provided traces that trigger this issue, it can correctly mark those traces as ERRONEOUS 90% of the time.

4.4.4 RQ2: State-of-the-art Comparison

Figure 4.8 presents a comparison of the performance of Mithra against AR-SI. The median precision, recall, and accuracy of AR-SI are 62.2%, 39.0%, and 57.8% respectively, compared to Mithra's 74.7%, 56.0%, and 69.3%. Using a Mann-Whitney U test ($\alpha = 0.05$) we demonstrate that Mithra achieves significantly higher precision, recall, and accuracy compared to AR-SI. That

[7] `http://ardupilot.org/copter/docs/mission-command-list.html#loiter-time`

Table 4.1: A comparison of Mithra's performance when the output clusters of one of its steps is used to construct the classifier in terms of precision, recall, and accuracy, reported by their median and interquartile range across 20 seeds. Using a one-sided Mann-Whitney U test [189], we show that both Behavioral Clusters and Subclusters have significantly higher precision and accuracy, and lower recall than Preclusters ($\alpha = 0.01$). We are unable to find a significant difference between Subclusters and Behavioral Clusters.

	Preclusters		**Subclusters**		**Behavioral Clusters**	
	Median	IQR	Median	IQR	Median	IQR
Precision	0.52	0.01	0.72	0.06	0.75	0.06
Recall	0.96	0.03	0.59	0.04	0.56	0.04
Accuracy	0.54	0.02	0.68	0.02	0.69	0.02

is, Mithra detects a greater number of erroneous traces and does so with higher confidence.

We additionally use the intra-class correlation coefficient ICC$(3, 1)$ [161] to measure the reliability of Mithra and AR-SI across 20 seeds. This metric measures the consistency of a model in assigning the same label to a given trace across different seeds, and takes a value between zero and one; one being perfect reliability, and zero the complete absence of reliability. We find that Mithra demonstrates a "good" reliability of 0.840, whereas AR-SI exhibits a "poor" reliability of 0.349. Intuitively, this result shows that Mithra is more likely to assign the same label to a given trace regardless of the seed used during training.

4.4.5 RQ3: Conceptual Validation

Each of the three steps of Mithra's clustering approach is designed to improve the accuracy of its detected clusters while supporting scalability. To evaluate the individual impact of those steps, we used the output produced by each step (i.e., preclusters, subclusters, and behavioral clusters) as input to oracle querying, which we then used to measure the performance of each step (Table 4.1).

Using the outputs of either the second or third step of our approach (i.e., subclusters and behavioral clusters) to produce a classifier results in significantly higher precision and accuracy ($\alpha = 0.05$) than a classifier constructed using the output of only the first step (i.e., preclusters). This finding demonstrates that solely using Dynamic Time Warping on low-resolution data is insufficient on its own for precisely detecting behavioral patterns.

We were unable to show a significant difference in performance between using subclusters and behavioral clusters. Recall, however, that the intention behind Mithra's third step is not to improve functional performance, but rather to effectively reduce the number of reported clusters by combining clusters that represent the same behavior. On average, Mithra identifies 21 subclusters for each command, which it reduces to an average of 5.5 behavioral clusters after its merging step. By merging non-unique clusters, we reduce the cost of oracle querying by decreasing the number of expensive DTW distance calculations. Furthermore, reporting fewer clusters may ultimately aid user comprehension of the discovered behaviors and thus provide higher confidence in the output of the technique. However, non-unique clusters do not impact Mithra's performance since oracle querying is independent of cluster uniqueness. Our results

provide empirical evidence that the process of merging clusters is indeed effective at reducing the number of clusters, and does not have any significant impact on overall performance, thereby indicating that information is preserved.

To investigate the importance of preclustering, we applied step 2 of Mithra's approach in isolation to the original training traces. The resulting classifier obtained a median precision, recall, and accuracy of 53.2%, 90.9% and 55.4%, respectively, providing evidence that preclustering of low-resolution traces with DTW before subclustering results in significantly higher precision and accuracy ($\alpha = 0.01$).

4.4.6 RQ4: Time

Our approach for automatically generating CPS oracles requires an up-front training stage, whereas AR-SI can simply be applied to evaluation traces without training. Although Mithra's training can take several hours to complete, depending on the size of the training data, that cost only needs to be paid once and can be amortized. For our experiments, Mithra's training took 4 hours and 45 minutes to complete and was spread across 30 threads. However, by storing and reusing computed distance matrices, Mithra's training time for subsequent seeds was reduced to an average of 29.59 minutes. AR-SI's cost of labeling a single query trace is relatively expensive, since it repeatedly optimizes a number of parameters for every datapoint in the trace. In our experiments, on average, it took 27.65 minutes for AR-SI to label all evaluation traces using 30 threads (i.e., each trace took approximately 107 thread-seconds). For Mithra, it took an average of 2.79 minutes to label all evaluation traces using 30 threads (i.e., each trace took approximately 11 thread-seconds). Using an independent samples t-test, we show that querying Mithra is significantly ($p < 0.001$) faster than AR-SI.

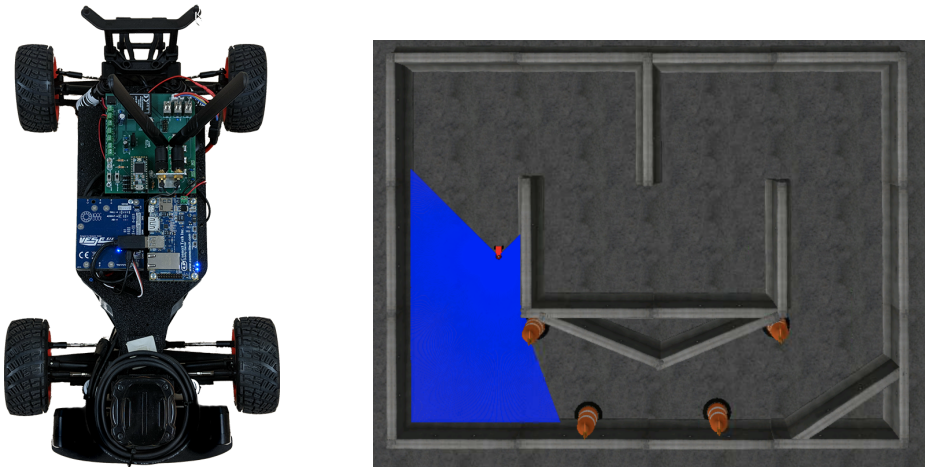
Overall, Mithra does require an upfront training cost that AR-SI does not; given a trained model, oracle querying for Mithra is approximately 10X faster than AR-SI.

4.4.7 Wider Applicability

To show that Mithra is not limited to a single system (i.e., ARDUPILOT), we demonstrated Mithra on the F1/10 platform [217], shown in Figure 4.9. F1/10 is an open-source, autonomous racing cyber-physical platform, one tenth of the size of a real Formula 1 racing car, that is designed to be used as a testbed for research and education. We chose F1/10 as an additional case study to demonstrate the applicability of Mithra to a system built on top of the Robot Operating System [229], the most popular robotics development platform, sometimes referred to as the "Linux of Robotics" [279].

In this experiment, we used Mithra to learn an oracle for the wall-following command of F1/10,[8] in which the vehicle uses its sensors to complete laps around the race track without crashing. The wall-following command takes a single parameter that specifies whether the vehicle should follow the inside or outside walls of the track. The vehicle will indefinitely complete laps around the track in a counter-clockwise direction, remaining close to desired wall, until instructed to stop by the user. Since "missions" for this system consist of a single command

[8] `https://github.com/linklab-uva/f1tenth_gtc_tutorial`



(a) (b)

Figure 4.9: (a) The F1/10 vehicle; one tenth of the size of a real Formula 1 race car. (b) A simulated race track with four obstacle cones (orange), the F1/10 vehicle (red), and the range covered by the vehicle's sensors (blue). The vehicle follows the inside or outside walls to navigate through the track counter-clockwise, and avoids obstacles.

of indefinite duration, we imposed a wall-clock time limit when collecting traces. These traces consist of seven state variables, describing the vehicle's position and orientation at each point of observation.

We assessed Mithra on F1/10 using a similar approach to our evaluation on ARDUPILOT, outlined in Section 4.4.2, by constructing a benchmark. We used simulation to construct a training dataset of 75 unlabeled traces, covering both inside and outside wall-following behaviors. Note that we collect substantially fewer training traces for F1/10 compared to ARDUPILOT since the latter has a greater set of commands and parameters. To construct an evaluation dataset, we first automatically injected 234 faults into the F1/10 source code using four mutation operators: Wrong Arithmetic Operation, Wrong Value Assigned to a Variable, Missing Parentheses, and Wrong Logic Clause. We used artificial faults for evaluation since F1/10 does not have a rich enough development history to extract historical faults. After running the command with both parameters on the syntactically valid, non-crashing bugs and collecting the traces, we manually identified the mutants that led to failure (i.e., crashing into obstacles). We identified 153 mutants and produce 261 faulty traces. To ensure a balanced evaluation dataset, we collected an additional 261 traces using the unmodified F1/10 system.

We ran Mithra with rarity threshold $\rho = 5$, maximum number of clusters $k_{max} = 15$, and without feature selection, and repeat the experiment with 20 seeds. Mithra reached its highest median accuracy (81.0%) when $\theta = 1$, with median precision and recall of 84.6% and 74.9%, respectively. By comparison, AR-SI achieved its highest median accuracy (51.3%), with a median precision and recall of 51.7% and 37.1%, respectively, when $p = 10$.

The high performance of Mithra on F1/10 may be explained by how erroneous behaviors in this system manifest. In most cases, the vehicle misbehaves smoothly, and does not necessarily show sudden, unexpected changes; rather, it slowly navigates along the wrong path. Mithra

detects that the behavior does not match previously identified behavioral clusters. In contrast, AR-SI only detects erroneous behaviors that involve abrupt changes, which may explain why AR-SI performs poorly on F1/10.

Overall, these results demonstrate the wider applicability of Mithra by showing that Mithra can be successfully applied to another system (i.e., F1/10).

4.5 Limitations and Threats

Limitations Our approach, like others, treats anomalous behavior as erroneous, and common erroneous behavior as correct [29, 65, 88, 136, 173, 216, 288]. However, anomalous behavior also includes corner cases and rare behaviors that are not observed during training, which are not necessarily erroneous, and erroneous behavior can be observed in the training data. Even though reporting the anomalous-yet-correct behaviors as erroneous is not ideal, it can inform the developers of under-tested functionality. In addition, most systems typically perform as expected [85], as it is easier for developers to detect and debug an erroneous behavior that is observed frequently.

Overall, the performance of our approach depends on its training data, a limitation it shares with other dynamic model learning techniques [29, 88, 173, 216]. If the provided traces do not provide sufficient coverage of the unique behaviors of the robot, our approach will fail to identify those behaviors. However, generating a diverse set of training data is an orthogonal problem we leave to future work.

Our approach assumes sequential execution of commands and cannot handle asynchronous or concurrent executions. We leave including such traces in our approach to future work.

Threats to Validity In theory, our approach is applicable to any CPS that logs its telemetry data. However, we only evaluated on two instances of such systems. We picked ARDUPILOT as a fairly complex and highly popular system that is widely used as a representative of real CPSs in prior work [18, 124, 179, 274, 296], and we pickedf F1/10 as system built on top of the popular Robot Operating System [229].

In many CPSs, executing faulty lines and triggering a bug does not guarantee that the bug will manifest. However, many of our bugs are associated with a bug report on ARDUPILOT's issue tracker and describe missions that manifest the bug. We created mission templates based on the bug reports and our own understanding of the bugs, and generated random missions from those templates. The mission templates are a source of internal validity.

As the source code of AR-SI was not available to us, we implemented our own version of AR-SI based on the description provided in the paper [124]. Our implementation of AR-SI represents a potential threat to internal validity. We release our implementation of AR-SI as part of our replication package.

Although Mithra is agnostic to the source of its traces and can be applied to field traces, we do not evaluate Mithra on field traces and leave that for future work.

4.6 Summary and Future Work

To summarize, in this chapter, I introduced Mithra, an automated tool that demonstrates a three-step multivariate time series clustering approach as an effective means of generating oracles for cyberphysical systems. As part of our evaluation on widely used robotics platform of ARDUPILOT, we showed that Mithra identifies a higher number of faulty executions than AR-SI, a state-of-the-art oracle generation technique for CPSs, and does so with a higher level of confidence. We showed that Mithra is generally more reliable and may be used to provide an oracle for automated, simulation-based testing as part of a continuous integration and deployment workflow.

There are a number of improvements that can be added to Mithra to make it more accurate and usable for robotic and CPSs. In the current form, the clusters found by Mithra cannot be evolved to incorporate newly collected data. For example, if a new feature is added to the system, the oracle (i.e., behavioral clusters) need to be recomputed from scratch to take the modified behavior of the system into account. This limitation can result in high cost of using Mithra in practice on frequently evolving systems. Future studies can propose techniques to reuse the previously identified clusters, and evolve them based on newly collected data.

Another avenue for future work can be directed towards involving the users in some sort of verification of the behavioral clusters to improve the accuracy of the oracles. For instance, a few representative traces from each identified clusters can be presented to the user. The user can guide Mithra on whether a behavioral cluster conforms to their expectation, or modifications such as breaking the cluster into smaller clusters should be performed. Additionally, the users can provide insight on whether a cluster is presenting CORRECT or ERRONEOUS behavior, which can address the limitation regarding frequently-observed erroneous and rarely-observed correct behaviors.

Mithra in its current form completely treats the system as a black-box, and assumes no information about the SUT, except for the optional definition of dynamically computed dimensions presented in Section 4.3.4. In order to automatically identify relationships between different dimensions, Mithra needs to know the units for each dimension. For example, Mithra can automatically compare dimension A and B, if it is aware that both of these dimensions are measuring speed. However, it will not compare A and B to each other if one is measuring speed while the other is measuring angle. As a result, one solution to replace definitions for dynamically computed dimensions is to expect minimal annotations from the user that specify the units for each dimension, and use them to automatically infer relationships across dimensions. Additionally, we can take advantage of automated, white-box analysis techniques to learn these units without requiring the users to manually specify them [219, 220].

Finally, in this chapter, I limited the scope of the work to different behaviors of the system under a single configuration setting, and in a fixed environment. In future, we can propose ideas on how to effectively encode the configuration settings, and the simulated environment for each trace, so that our clusters will be reflective of different behaviors the SUT can have under different setups. For example, *distance to the closest object* can be a variable of the traces that in some way encodes the environment.

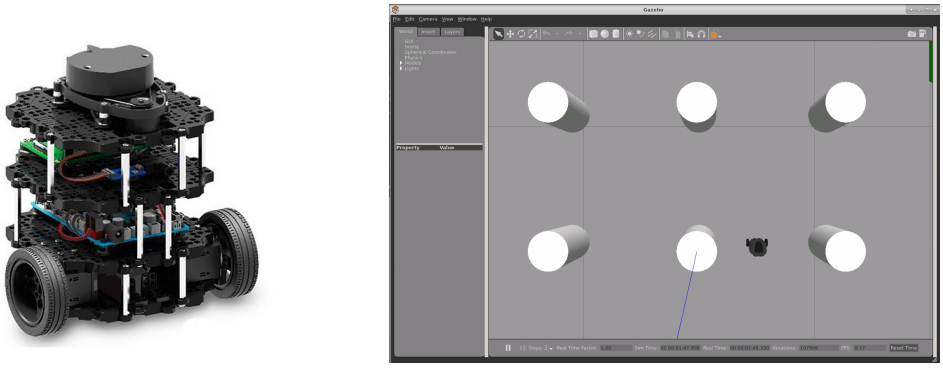
Chapter 5

Test Input Evaluation and Generation

Test inputs are essential to perform any sorts of software testing. In fact, the quality of software testing, how well the testing process can find all the flaws in the system, is significantly affected by the quality of the test inputs provided to it [156]. Therefore, it is important to create test inputs that can expose the SUT to different scenarios, and shed light on its flaws.

Generating high quality test inputs for robotic and CPSs is extremely challenging as the input space in these systems are massive (e.g., system's configuration, environment, commands and parameters), making it infeasible to be exhaustively covered [232]. Even though the challenge of a vast input space is not specific to robotic and CPSs, it can be more manageable in non-robotics, software systems. In software systems (e.g., a web application), well-defined interfaces control and limit the range of inputs that can be received from external sources (e.g., users). However, a robot or CPS receives its inputs from the real, physical world, and any situation or scenario that can arise in the real world is part of the system's input space, which in many cases cannot be controlled or limited. Therefore, searching in the input space for automatically constructing meaningful test cases is one of the most important needs in the CPS testing domain [110]. In fact, handling *unpredictable corner cases* is one of the main challenges of testing these systems as described by robotics practitioners in our qualitative study of Section 3.1 [20].

As a result, an automated testing pipeline for robotic and CPSs such as the one presented in Figure 1.1 should include an automated means of generating test inputs that efficiently and effectively samples the vast input space to increase the testing framework's ability in revealing faults in the system [156]. Prior studies have proposed a number of automated test input generation approaches to address this problem [110, 122, 197, 205, 268, 269, 277]. However, many of these approaches require pre-defined artifacts and models of the system (e.g., Simulink and MATLAB models) [122, 197, 205, 277], which are difficult and error-prone to be specified [110], and may not be available for many non-safety-critical robotic and CPSs [122]. Other approaches specifically focus on autonomous driving applications with a set of assumptions and requirements (e.g., e.g., the definition of safe driving that includes traffic laws) that may not be easily extendable to other systems [110, 268, 269]. In this chapter, I study and evaluate different characteristics of the test inputs that impact their quality, and propose an automated, search-based test generation approach to generate high-quality test inputs without requiring pre-defined models or artifacts of the system. I conduct these studies on a case study mobile robot, presented in Section 5.1, to test ROS's navigation planner subsystem, used by thousands of users. Similar to Chapter 3 and



(a) (b)

Figure 5.1: (a) The TURTLEBOT3 mobile robot. (b) A simulated environment with six obstacle pillars (white), and the TURTLEBOT3 robot (black) in Gazebo.

4 my focus in this chapter is on fully-automated whole-system tests, which are performed using software-in-the-loop simulation.

Overall, to automatically generate test inputs for a robotic system that can be executed in simulation, we require the following items:

1. A way to specify test input scenarios, which include the specification of the simulation environment and the test mission to be performed, and a way to automatically generate the necessary files and models to construct the specified scenario in simulation.
2. A metric that can measure the quality of test inputs with respect to their effectiveness in revealing faults.
3. An approach to automatically generate test inputs with higher quality.

In Section 5.2, I present GzScenic, a tool I created to automatically translate test scenarios provided in the Scenic domain-specific language [97] to valid simulation scenes and missions, which satisfies the first requirement. I address the second requirement from the list above, by 1) investigating the reliability of test outcome with respect to determinism in the test executions for different test inputs, as it affects the quality of tests [53, 93, 127, 170, 184] and how we evaluate them, and 2) using the mutation score [76] as the ground-truth measure for the effectiveness of test inputs [141, 222] to evaluate different coverage-based test input quality metrics (Section 5.3). Finally, in Section 5.4, to automatically generate high-quality test inputs, I propose an evolutionary-based test generation approach with a fitness function that is based on *scenario coverage*, where the test generation or selection approach focuses on maximizing the diversity and effectiveness of the scenarios presented to the system under test [102, 150, 202, 233, 281].

5.1 Case Study

In this work, I set out to examine the test input characteristics of mobile robots specifically with regards to the path finding and planner subsystems [211]. Mobile robots include a wide variety of systems that are used for a wide range of purposes. To name a few TURTLEBOT3 [10], FETCH [278], HUSKY [5], and PR2 [9] are all well-known mobile robots built on top of ROS,

and are used for a variety of purposes.

`navfn` and `base_local_planner` are packages in the ROS navigation stack[1] that are commonly used by mobile robots that are developed on top of ROS. The `navfn` package provides a fast interpolated navigation function that can be used to create plans for a mobile base, and uses Dijkstra's algorithm. The `base_local_planner` package provides implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation on a plane. Given a plan to follow and a costmap, the controller produces velocity commands to send to a mobile base. In other words, these two packages can be used in a mobile robot to determine a path between the source and the destination, and compute the appropriate velocity commands to be used by the actuators.

Since these packages provide essential functionality for the mobile robots, and are highly popular [160], it is extremely important to test them under different scenarios, and investigate their robustness. In this work, I use TURTLEBOT3 robot as an example system that uses the aforementioned packages for navigation.

TURTLEBOT3 [10], shown in Figure 5.1a, is a programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping [25, 32, 227]. TURTLEBOT3 can be customized into various ways depending on how you reconstruct the mechanical parts and use optional parts such as the computer and sensor. TURTLEBOT3 uses simultaneous localization and mapping (SLAM) [174] algorithms to build a map, which it uses alongside ROS navigation stack to drive around a room. It can be simulated using the popular Gazebo simulator [159] (Figure 5.1b).

5.2 Simulation Scenario Construction

```
ego = Car

spot = OrientedPoint on
    visible curb
badAngle = Uniform(1.0, -1.0)
    * Range(10, 20) deg
parkedCar = Car left of (spot
    offset by -0.5 @ 0), facing
     badAngle relative to
    roadDirection
```

(a)

(b)

Figure 5.2: (a) An exemplary scenario description, written in the Scenic language, detailing a scene that contains a badly parked car, and (b) A scene that was generated by Scenic according to the scenario above using the GTA V engine [97].

In order to test a single TURTLEBOT3 in simulation, we need to define test inputs (i.e.,

[1] `https://github.com/ros-planning/navigation`

scenarios) that specify several key elements:

1. The environment where the simulation takes place (a.k.a., scene). A scene describes all the elements in a simulation, including its objects, sensors, light sources, etc.
2. The initial position and orientation of the robot.
3. The set of waypoints (i.e., goals) that should be visited by the robot to perform the navigation mission.

In addition to the aforementioned items, we need to specify the set of configuration options selected to configure the robot under test. Even though exploring configuration space is a challenging and important part of testing robotic systems [151], it is beyond the scope of my work. Therefore, I only focus on generating test inputs for the default configuration of the system, and leave efficient exploration of the configuration space to future work.

As mentioned in Section 3.3, manually generating the scenes and missions for testing in simulation can be time consuming and difficult [21]. In recent years, researchers have proposed tools and domain-specific languages (DSLs) to facilitate the construction of testing scenarios [97, 158, 187]. One of the most prominent such DSLs is Scenic [97], a language designed for creating simulation scenarios for autonomous vehicles. Using Scenic, users can describe a scenario of interest for the SUT, which is automatically parsed by the Scenic tool to generate a plausible scene and mission that satisfy the user-specified constraints of that scenario. The generated scene and mission are then executed in the supported simulators to execute the test. Figure 5.2 shows an example scenario that is realized in the GTA V [236] simulator. Scenic requires a pre-defined set of models that define everything specific to a particular simulator and SUT. For example, for an autonomous driving system Scenic models need to describe entities such as `Car`, `Road`, and `Pedestrian`, and how they should be rendered in a specific simulator (e.g., CARLA).

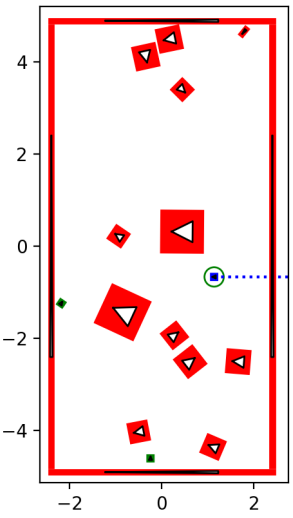
Although Scenic provides a powerful language and tool that simplifies the process of creating and running simulated test scenarios, it only supports domain-specific simulators in the autonomous vehicle sector, and is not compatible with Gazebo; the most popular, general-purpose robotic simulator [159]. Since Gazebo is a general-purpose simulator that is used in a wide range of domains, it is nearly impossible to pre-define Scenic models that describe the entities required for simulation of all systems in different sectors. For example, an agricultural robot requires modeling of entities such as plants and tractors, whereas a warehouse robot requires modeling of the shelves, boxes, and rooms. In comparison, defining these models for a domain-specific simulator such as GTA V and CARLA requires a one-time investment since most of the entities that can be simulated and included in the scenarios are shared among all systems that use these simulators. For example, if models are produced for CARLA in order to test a given system, those same models may be reused in another system with minimal effort.

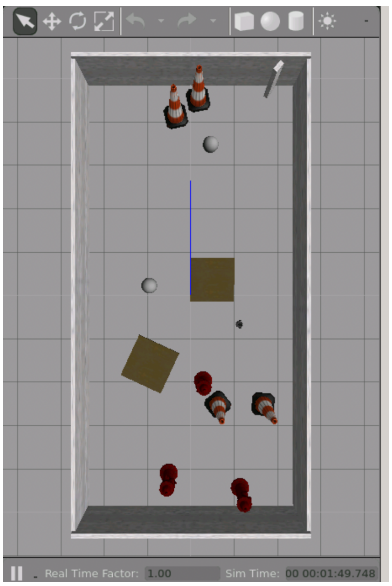
To be able to automatically generate simulation scenes in Gazebo from a scenario provided in Scenic's DSL, for TURTLEBOT3 or any other ROS systems, I created the GzScenic tool. Using GzScenic, developers can specify their desired testing scenarios in Scenic's DSL without the need to manually pre-define their models in Scenic, and automatically generate complex scenes that satisfy the constraints of their scenario. GzScenic automatically transfers the generated scenes to Gazebo without the need for manual translation. Furthermore, to support test automation for mission-based robots, GzScenic can synthesize mission items (e.g., waypoints, action locations, the initial position of a robot) as part of a test scenario.

```
ego = Turtlebot3
create_room(5,10)

for _ in range(2):
  CafeTable
for _ in range(4):
  ConstructionCone
for _ in range(3):
  FireHydrant
for _ in range(2):
  Ball
GreyWall

wp_region = Region(0 @ 0,
    0, 4.5, 9.5)
Waypoint in wp_region
Waypoint in wp_region
```





(a) (b) (c)

Figure 5.3: (a) An example scenario written in Scenic language for TURTLEBOT3 randomly placing 12 objects and 2 waypoints in the scene, (b) the 2D plot of the generated scene, and (c) the simulation of the generated scene in Gazebo.

Overall, GzScenic allows the users to simply specify a list of models they intend to use in the simulation, and it automatically turns these models into models that are interpretable by Scenic. Using these models, Scenic generates a scene from the scenario, which later on is automatically converted to Gazebo models by GzScenic [22]. Figure 5.3 illustrates an example of a Gazebo scenario for TURTLEBOT3 that is automatically generated from the presented (simplified) Scenic scenario. I use GzScenic in the following sections to execute automatically or manually generated test scenarios for TURTLEBOT3 in simulation.

5.3 Test Input Quality Metrics

In the previous section, I presented a tool that allows for automated and easy construction of simulation scenarios, which as discussed earlier, is one of the pieces required to develop an automated test generation approach. Besides the ability to automatically construct simulation scenarios, we need to develop an understanding of what makes a test input or a test suite more effective (i.e., high quality), and what characteristics in the test inputs make them more reliable. This understanding allows us to be able to evaluate different test inputs, and propose an effective automated test input generation approach for robotic and CPSs.

In this work, I define the quality or effectiveness of a test suite as its ability to reveal faults in the system [156]. One such metric is the *mutation score* [76], which is calculated by executing a test suite on a set of mutated programs (i.e., programs that are injected with faults), and measuring the number of these executions that result in different outputs than running the same test suite

on the original, unmutated program. As described in Section 2.2, mutation score can be used to measure the effectiveness of a test set in terms of its ability to detect faults [141], and higher mutation score is proven to improve the fault detection significantly [222].

However, computing the mutation score is extremely expensive and in many cases infeasible as it requires the execution of test suites on many variations of the program (i.e., mutants), which requires a lot of time and resources depending on the SUT. For example, let us assume that each test for our case study system, TURTLEBOT3, takes 5 minutes on average, and we would like to evaluate a test suites with 10 test cases. For a set of 200 mutants, it is going to take $10 \times 200 \times 5 = 10000$ mins $= 166.66$ hours to calculate the mutation score. The severity of this problem becomes even more considerable when multiple test suites are being evaluated and compared against each other, possibly in the process of automated test generation.

Instead of using mutation score to assess the quality of test suites, different coverage-based metrics (e.g., function coverage, statement coverage, branch coverage) have been proposed as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [131, 185, 230, 259]. Automated test generation tools and approaches for conventional software commonly use coverage metrics as a mean of evaluating their test suites, and attempt generating test suites that maximize the coverage metrics [95, 177, 223, 287].

Despite the popularity of coverage metrics for test suite evaluation, these metrics are shown to be poor indicators of fault finding effectiveness [57, 96, 107, 125, 137]. Marick explains that coverage metrics are only able to tell us how the code that exists has been exercised, but they cannot tell us how code that ought to exist would have been exercised (faults of commission vs. faults of omission) [190, 191], and missing logic is one of the main reasons behind coverage metrics' poor performance [125].

I hypothesize that the problems of coverage metrics are even more severe in robotic and CPSs. First of all, many potential faults in these systems can arise when the system has no components to handle certain situations [123]. For example, if the navigation planner of TURTLEBOT3 has no logic for handling situations where the robot is stuck in a corner, then the coverage metric will not provide any information regarding the failures that arise in that specific situation. Second, the robotic systems include a control loop that periodically (with high frequency) executes the control logic of the program. Therefore, most of the statements of the program are executed during the overall execution of the test. Finally, as mentioned in Section 3.2, triggering the faults in these systems do not always result in them being manifested. In other words, scenarios that execute the faulty code may not result in a faulty behavior unless a specific condition arises. This phenomena is called coincidental correctness [56], and reduces the effectiveness of testing [33, 128, 194, 195, 273]. Therefore, generating tests with high code coverage may not necessarily lead to the manifestation of more failures in the system.

In addition to the these challenges, the non-deterministic nature of robotic and CPSs can have serious ramifications for testing and test input generation [93, 184]. With non-deterministic systems and software, we can run the exact same test case (i.e., with the exact same test inputs under the exact same test preconditions) multiple times and get different results (i.e., different test outputs and test postconditions). First of all, testing non-deterministic systems require a more complex oracle as mentioned in Chapter 4, since a single behavior is not truly representative of all possible (or accepted) outcomes of a test. Second, a single execution of the test inputs is not sufficient to mark the test as passing or failing. This concept is known as *flaky tests* that can non-

deterministically pass or fail when run on the same version of the program [43]. Prior studies have investigated flaky tests in conventional software systems extensively, and have introduced approaches to automatically identify flaky tests in a test suite [43, 94, 115, 118, 165, 166, 167].

Even though there are strong indications of the existence of non-determinism in robotic and CPSs [93, 184], to the best of my knowledge no empirical data is available on how severe the non-determinism is in these systems, and how it varies among different test inputs. Identifying the severity of this issue can 1) provide insight on how reliable the test executions on these systems are, and how the test inputs should be evaluated, 2) inform our automated test generation approach on generating test inputs with higher reliability, and 3) lay the ground work for future studies on resolving or controlling the non-determinism in the system.

As a result I set out to answer the following research questions:

- **RQ1:** *How severe is non-determinism in robotic software?*
- **RQ2:** *How effective are coverage metrics in evaluation of test suites for robotic software?*

As mentioned earlier, I investigate the ROS navigation function and local planner subsystems using TURTLEBOT3 robot as a subsystem that is commonly used by systems developed on top of ROS.

5.3.1 Methodology and Data Collection

To answer the aforementioned research questions, I use the methodology similar to the technique taken by prior work [57, 137] using mutation score as ground-truth. After creating a set of test inputs and executing them on the original, unmutated program multiple times (because of possible non-determinism in the executions), I create a set of mutants from the original program, and run the test cases on each mutant multiple times. I compute the mutation score for each test input by determining whether the output or performance of the robot on the mutant is statistically different from the behavior of the system with the original program. Note that because of non-deterministic executions, I need to use statistical methods to compare the two sets of executions; one on the original program, and another on the mutant. I later use the data collected over these executions to study the severity of non-determinism (RQ1), and the correlation between mutation score and multiple coverage metrics (RQ2).

Test inputs I first need to create a diverse set of test inputs that expose the system to different navigation scenarios. For this purpose, I manually created 9 scenarios written in Scenic language [97], which I used to create 16 test inputs (i.e., simulation scene, the robot's initial position, and the navigation mission) using GzScenic (Section 5.2). Note that multiple scenes can be generated from a single scenario when the scenario includes random positioning of objects (7 out of 9 scenario include randomness). I manually created these scenarios to ensure they include high levels of diversity. For instance, I included both scenarios where the system navigates on a mostly empty plane, and the scenario where the robot has to navigate through a crowded space. I also included 3 purely random scenarios that slightly differ from each other in the number and type of objects they include to mitigate the threat of introducing bias to the data.

Original runs To observe possible non-determinism in the system, I ran each test input 20 times in simulation with a timeout of 5 minutes, and collected the following data:

- The status of test completion: Whether the test execution ended normally (i.e., `OK`), or resulted in a `TIMEOUT`, `ERROR`, or system `CRASH`.
- Whether the robot hit obstacles.
- The maximum localization error: the localization error (LE) refers to the difference between the location where the robot thinks it is (determined by the localization subsystem that is SLAM in TUTLEBOT3), and where it actually is according to the ground truth provided by the simulator. I take the maximum error as it presents the worst localization that the robot received during the mission, which directly impacts the navigation function.
- The robot's closest distance to the specified waypoints of the mission.

Since there are no well-established, accurate oracles for robotic systems, I focus on the data points that are important in determining whether a navigation mission has been successfully completed [140].

Table 5.1: The mutation operators used to generate a set of mutants.

Type	Description
aar	Array reference fr array reference replacement
aor	Arithmetic operator replacement
crp	Constant replacement
ror	Relational operator replacement
svr	Scalar variable replacement
uoi	Unary operator insertion

Mutants To generate mutants of the original program that affect navigation function and the local planner, I use six mutation operators [215] presented in Table 5.1 on the two `.cpp` files in the `navfn` and `local_base_planner` packages that control parts of the main navigation logic (total of 1,371 LOC). I used Comby [271], a tool for searching and changing code structure, to apply the mutations to the code. Overall, applying all possible mutations on the two `.cpp` files resulted in 1632 mutants. However, not all of these mutants are syntactically valid and compilable. Therefore, after removing all not compilable mutants I ended up with 1146 valid mutants. In order to control the scale of these experiments, I randomly selected 200 mutants from the set of 1146 valid mutants.

Mutant runs Because of possible non-determinism in the system, I execute each test input 8 times on every mutant. The reason behind a lower number of executions on the mutants compared to the original program is the high cost associated with this step of data collection. This is by far the most expensive part of the data collection and takes a significant time. In fact, each round of running all 16 test inputs on all 200 mutants takes more than 16 hours with 10 executions running in parallel. On every execution, I collect the same data as the one described for original runs.

In order to mark a mutant as detected or *killed* by a test case, we need to determine whether the output of running the test on the mutant is different than the output of running the same test on the original, not mutated program. However, as mentioned earlier, in these systems we need to deal with non-deterministic outcomes, and therefore, we have to compare two distributions against each other. For this purpose, I use the Kolmogorov–Smirnov (K-S) test, which is a nonparametric test of the equality of continuous one-dimensional probability distributions that can be used to compare two samples (two-sample K–S test) [196]. A p-value smaller than 0.05 shows that the two samples do not come from the same distribution. Another statistical test that can be used in this context is the Mann-Whitney test [189]. However, the K-S test is sensitive to any differences in the two distributions. Substantial differences in shape, spread or median will result in a small p-value. In contrast, the Mann-Whitney test is mostly sensitive to changes in the median, and as a result I use the K-S test in this study.

In order to mark a mutant as *killed* by a test case, at least one of the following conditions should be satisfied:

- At least one of the executions of the test on the mutant did not terminate normally.
- At least one of the executions of the test on the mutant resulted in hitting obstacles while none of the original executions showed such behavior.
- The p-value of K-S test on the maximum localization error of all test executions on the original program and the mutant be smaller than 0.05.
- The p-value of K-S test on the maximum closest distance to the waypoints of all test executions on the original program and the mutant be smaller than 0.05.

5.3.2 (RQ1) Non-determinism in Robotic Software

Many software systems (e.g., distributed systems, CPSs, embedded systems) exhibit a level of non-determinism in their behavior, meaning that running the exact same inputs under the exact same conditions may result in different outputs and behaviors [53, 93, 127, 170, 184]. Non-determinism can either occur when there is no theoretical way of predetermining the system's exact behavior (i.e., actual non-determinism), such as behavior that is determined by quantum physics, or it can occur when there is no practical way for the tester (or test oracle) to predetermine the system's exact behavior (i.e., apparent non-determinism) [93]. Non-determinism in the system can arise from many different sources [93] such as:

- Physical non-determinism: due to the nature of physical reality.
- Emergent non-determinism: due to integration of subsystems into systems.
- Concurrent non-determinism: due to system-internal and -external concurrency.
- Exceptional non-determinism: due to fault and failure behavior.

Since robotic and CPSs interact with the physical world, integrate many subsystems, and operate with high concurrency, they are highly susceptible to non-determinism.

As mentioned earlier, non-determinism in the system has serious ramifications for testing both in terms of complexity of the oracle (a single behavior cannot be truly representative of all possible (or accepted) outcomes of a test), and reliability of the test execution outcome (a single execution of the test inputs is not sufficient to mark the test as passing or failing). This concept

Table 5.2: The summary of 16 test scenarios executed 20 times each by the number of executions that terminated normally (i.e., ran OK) and resulted in hitting the obstacles, and the entropy, mean, and standard deviation of maximum localization error and maximum closest distance to the waypoints over 20 runs.

	Ran	Obst.	Max Localization Error			Max Closest Distance to WPs		
Test	OK	Hits	Entropy	Mean	Std	Entropy	Mean	Std
0	20	0	0.857	0.157	0.038	0.000	0.032	0.006
1	20	0	1.941	1.783	0.354	0.206	0.107	0.313
2	20	0	0.693	0.162	0.027	0.199	0.058	0.030
3	20	6	0.647	0.181	0.050	1.259	0.586	1.100
4	20	5	0.898	0.200	0.059	0.639	0.601	1.168
5	20	0	0.199	0.125	0.017	0.000	0.035	0.007
6	20	0	0.000	0.108	0.023	0.000	0.035	0.006
7	20	0	0.746	0.230	0.048	0.000	0.036	0.009
8	20	0	0.898	0.161	0.052	0.000	0.042	0.005
9	20	0	0.731	0.426	0.054	0.000	0.034	0.006
10	20	0	0.938	0.425	0.078	0.000	0.030	0.008
11	20	0	0.000	0.068	0.012	0.000	0.038	0.006
12	20	0	0.647	0.153	0.031	0.708	0.350	0.633
13	20	0	0.688	0.159	0.016	0.000	0.033	0.005
14	20	0	1.344	3.871	0.094	0.000	0.039	0.006
15	20	0	1.739	2.247	0.214	0.000	0.022	0.012

that is also known as *flaky tests* have been investigated extensively in conventional software systems [43, 94, 115, 118, 165, 166, 167]. However, to the best of my knowledge no empirical data is available on how severe the non-determinism is in robotic systems, and how it varies among different test inputs. Identifying the severity of this issue can 1) provide insight on how reliable the test executions on these systems are, 2) inform our automated test generation approach on generating test inputs with higher reliability, and 3) lay the ground work for future studies on resolving or controlling the non-determinism in the system. In this section, I investigate the non-determinism that is manifested through testing in robotic software by using the data of *Original runs* presented in Section 5.3.1, where I execute 16 test inputs, 20 times on the TURTLEBOT3 robot, and measure the variability of output behavior.

Results Figure 5.4 presents the density plot of the maximum localization error, and maximum closest distance to waypoints achieved among 20 runs for the first 5 test inputs. As shown in the figure, there is a high difference between variability of different test inputs. For example, tests #0 and #2 show a very low variability in both their localization error, and reaching the waypoints. However, test #1 shows a very high variability in the maximum localization error as the values range between 1.2 to 2.9 meters, and test #3 and #4 show high variability in the maximum closest distance to the waypoints.

Table 5.2 presents the data collected on all test inputs: 1) We can see that all test runs terminated normally in a timely manner. 2) There are 6 and 5 executions of tests #3 and #4 respectively

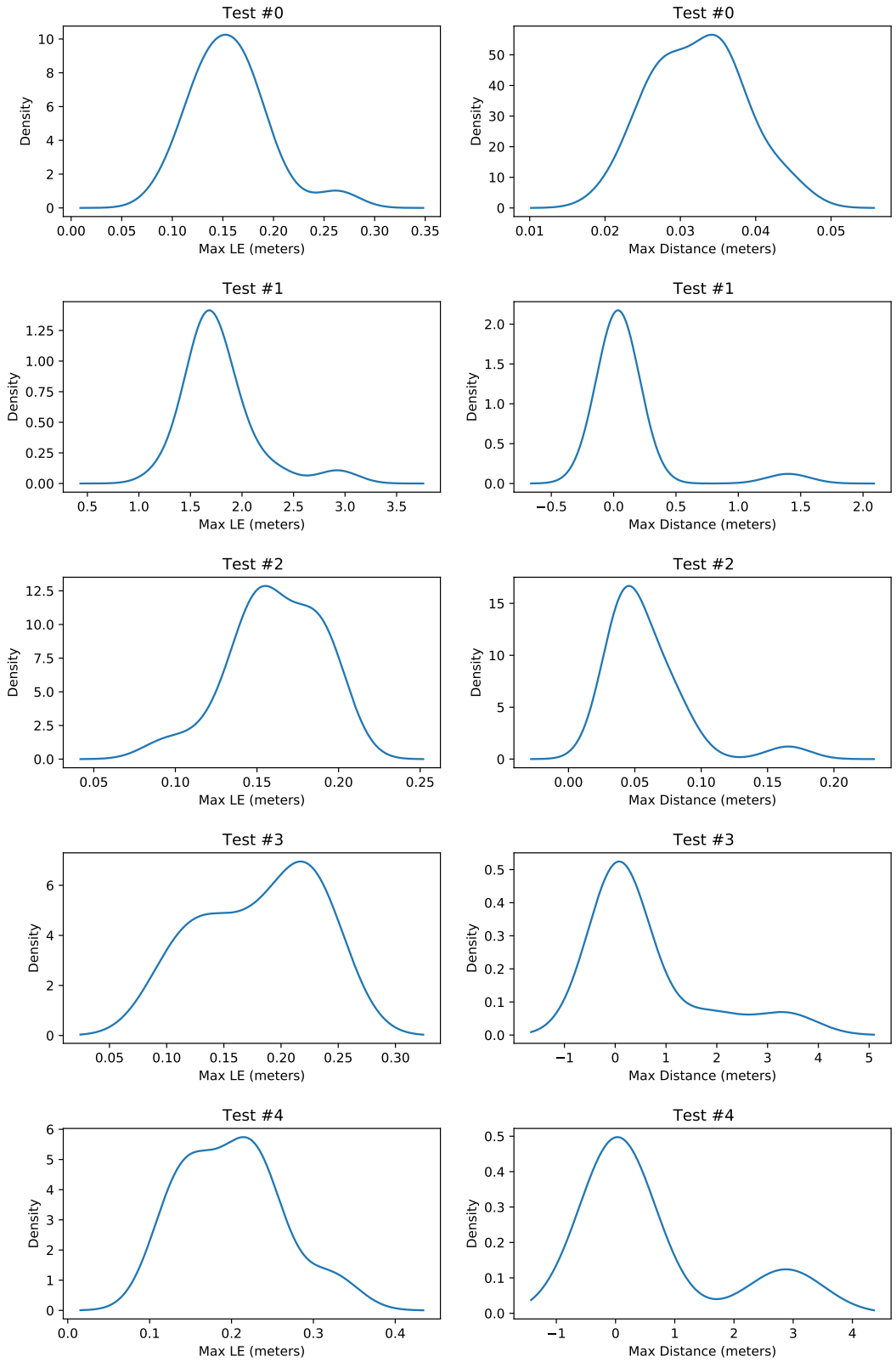


Figure 5.4: The density plots of the robot's maximum localization error (left diagram), and maximum closest distance to waypoints (right diagram) over 20 runs for the first 5 test inputs.

that result in hitting obstacles during the mission, which by itself shows a high degree of non-determinism on these tests. 3) The entropy and standard deviation of maximum localization error, and the maximum closest distance to waypoints highly vary among different tests. The higher entropy and std show higher variability and non-determinism. Even though there is a strong and significant correlation between entropy and std (Pearson correlation of 0.82 and 0.92 for maximum LE and maximum closest distance respectively with p-value<0.005), I consider std as a better measurement of variability in this context as it is more sensitive towards the absolute difference between observed values.

Overall, out of the 16 test cases, 9 have at least one non-zero entropy, and 4 have out of those 9 have the sum std of higher than 0.5 meter, which shows high rate of non-determinism in the test runs. This finding indicates that the test generation and evaluation techniques for these systems need to take non-determinism into account, and cannot rely on a single test execution to determine the outcome of the tests.

Additionally, by observing different levels of non-determinism among different scenarios, this empirical evidence suggests that there are characteristics of test scenarios that can result in higher or lower level of non-determinism among executions. In Section 5.4, I further analyze these characteristics, and present a metric that strongly correlates with the std measures of the test executions.

Sources of Non-determinism Our study shows high levels of non-determinism that exist among test executions in our case-study system. The non-determinism among executions can arise from many different sources such as noise in the sensor data, concurrency in the system, and timing of communication among subsystems. By identifying these sources, and investigating their impact on the non-determinism among test executions, we can provide methods and techniques to control and adjust them in simulation for a more reliable automated testing. In Section 5.6, I further discuss opportunities for future work in this area.

Based on our knowledge and understanding of the SUT, the simulator, and overall architecture of robotic systems, we can speculate over possible sources of non-determinism in these systems such as the ones stated earlier. While a number of these sources can easily be controlled in simulation (e.g., sensor noise), others require more advanced approaches to be adjusted and controlled (e.g., timing and synchronization among subsystems). As a preliminary study, I investigate three possible sources of non-determinism in our case-study system that can be manipulated in simulation, using the 16 test scenarios of Table 5.2:

1. **Sensor noise:** The level of noise in the sensor data can have a significant impact on the level of non-determinism among executions since it can result in different inputs being provided to the system in each run. In simulation, the noise in the sensor data is added and controlled by the simulator, and most simulators allow configuring the level of noise. For example, in Gazebo [159], which is used by our case-study system, the level of noise for each sensor can be specified as a distribution (e.g., Gaussian) with specified properties.

   To investigate the impact of sensor noise on non-determinism, I executed the test scenarios in simulation that is configured to add zero noise to the sensor data, and compared it against a set of executions using a simulator with high levels of sensor noise (Gaussian distribution with mean 0.0, and standard deviation of 0.2 and 1.0 for the `imu` and `laser`

sensors respectively). The executions with high sensor noise result in significantly higher standard deviation in maximum localization error compared to executions without sensor noise (mean of 1.14 vs. 0.07). They also result in higher standard deviation in the maximum closest distance to the waypoints, but the difference is not statistically significant (mean of 0.46 vs. 0.16). This data suggests that sensor noise can have a significant impact on the level of non-determinism among executions, and eliminating the sensor noise can result in more reliable test executions in simulation. However, removing the sensor noise all together does not guarantee deterministic executions, as there are other sources of non-determinism, and it can result in less realistic simulations, since in reality the sensor noise cannot be eliminated.

2. **Simulation speed:** The speed of simulation refers to the frequency at which the simulation time steps are advanced, and it can impact the behavior of the system as it determines the amount of time the system can take to react to the inputs. For example, if the robot has infinite time, it can finish all the computations required to make decisions and react to a set of sensor readings in time. However, in a sped-up simulation, the system may not be able to react to the inputs in time, which can result in non-deterministic behavior.

   To investigate the impact of simulation speed on non-determinism, I executed the test scenarios in slowed-down simulations (0.5 the speed of real-time), and compared it against sped-up simulations (1.5 the speed of real-time). I observed *no* significant difference in the level of non-determinism among these two sets of executions, and they both resulted in similar standard deviation measures in the maximum localization error (mean of 0.08 vs. 0.07 for slowed-down and sped-up simulations respectively) and in the maximum closest distance to the waypoints (mean of 0.17 for both sets). This data suggests that slowing down the simulation may not necessarily result in more deterministic, reliable executions.

3. **Random seed:** The simulator and the SUT both use random numbers for different operations such as generating noise for simulation, and many random optimization algorithms used by different subsystems. These random numbers can be a source of difference among executions of a test scenario. We can attempt to control consistency of the generated random numbers by specifying a constant random seed. However, we cannot fully control the consistency since the generated random numbers also depend on the timing and synchronization of executions.

   To investigate the impact of adding random seed on non-determinism, I executed the test scenarios on a modified version of the SUT, where a constant random seed is provided to both the simulator and the the localization and mapping subsystem (SLAM) that uses random numbers for optimization. I compare these executions with a set of executions where no random seed is provided, and `time` is used as the default seed that changes among executions. Surprisingly, no significant difference among these two sets of executions was observed, and they both resulted in similar standard deviation measures in the maximum localization error (mean of 0.10 vs. 0.07 for executions with random seed and no random seed respectively) and in the maximum closest distance to the waypoints (mean of 0.19 and 0.20 respectively).

These results present a preliminary investigation on three possible sources of non-determinism

among test executions in simulation for TURTLEBO3. However, more data on a larger sample of systems is required to truly investigate this matter. For example, future studies can further explore the relationship between sensor noise and non-determinism in test executions, and dive deeper on how the non-determinism grows as the sensor data becomes more noisy. As another example, future studies can investigate the impact of simulation speed on non-determinism in more complex and computationally intensive systems.

Additionally, a number of possible sources of non-determinism such as concurrency in the system, and the synchronization among subsystems require more advanced approaches and engineering to be manipulated. For example, by using mocks and recorded data, future studies may be able to create a controlled environment to test a particular subsystem in a more reliable and consistent fashion. Identifying these sources of non-determinism, and tools and methods to control them for testing robotic systems can result in higher adoption of simulation-based testing, as reproducibility is one of the main challenges that robotics developers face while testing their systems in simulation [21].

5.3.3 (RQ2) Coverage Metrics and Mutation Score

To evaluate the effectiveness of coverage metrics (e.g., line coverage and branch coverage) in evaluating test inputs with respect to their ability in revealing faults, I compare these metrics with mutation score [76], which is shown to be a good indicator of test quality [141, 222]. Similar methodology is used by prior work to evaluate study coverage metrics on other systems [57, 137]. To answer this research question, I use the data described in Section 5.3.1, which consists of 16 test inputs, each executed 20 times on the original program, and 8 times on the 200 automatically-generated mutants.

Results Table 5.3 presents the number of killed mutants by each test according to the aforementioned criteria. In addition to the number of killed mutants, Table 5.3 presents the number of killed and saved mutants categorized by whether the mutant executions were likely deterministic. In other words, whether the 8 test runs on the mutant resulted in similar behavior (i.e., low std), or they differed from each other and were non-deterministic.

Table 5.4 presents the line and branch coverage measurements for each test. Note that since we have 20 runs for each test, there are different ways to measure line and branch coverage. In Table 5.4, I present four different measurements: 1) the mean line or branch coverage among all executions, 2) the number of lines or branches mutually executed by all test runs, 3) the number of lines or branches executed by at least one run, and 4) the number of lines or branches executed by the majority of runs (i.e., more than 10 runs).

Table 5.4 also presents the Pearson product-moment correlation between each coverage measurement, and the number of killed mutants presented in Table 5.3. As shown, most of the coverage measurements have a low and insignificant correlation with the number of killed mutants. However, if we consider the union coverage, we find a high and significant (p-value<0.05) correlation for both the line coverage and the branch coverage. In other words, the line and branch coverage are not suitable metrics to indicate testing effectiveness unless we consider all lines and branches that are covered by at least one execution of the test.

Table 5.3: The summary of the total number of mutants killed out of 200 by each test case, and the number of mutants killed and saved by the test cases categorized by whether the test executions on the mutant were likely deterministic or non-deterministic.

Test	Total killed	Det. killed	Det. saved	Non-det. killed	Non-det saved
0	53	31	146	22	1
1	80	22	93	58	27
2	56	22	137	34	7
3	48	18	25	30	127
4	54	25	39	29	107
5	49	25	150	24	1
6	36	16	163	20	1
7	48	21	144	27	8
8	49	28	151	21	0
9	66	44	134	22	0
10	47	21	153	26	0
11	49	30	151	19	0
12	68	27	29	41	103
13	51	26	141	25	8
14	44	20	154	24	2
15	47	19	152	28	1

As expected, the sum std of tests have a high and significant correlation with the total number of non-deterministic mutant executions (coef=0.94 and p-value<0.05), which confirms that the more non-deterministic the tests are the lower confidence we can have in the outcome of the tests. If we only consider tests with low variability in mutant outcomes, where the total number of deterministically labeled mutants is higher than 150 (three-fourth of the total number of mutants), we observe a higher but insignificant correlation between the number of killed mutants and the mean line and branch coverage (coef=0.49 and 0.57, p-value>0.05).

Besides measuring the quality of individual test cases through mutation score, it is important to measure the mutation score on a test suite since two lower quality test cases that target different behaviors of the system, or different scenarios can result in higher total mutation score than two high quality tests that target the same behavior [95, 171]. For this purpose, I create 10 test suites, each consist of five randomly selected tests from our original 16 tests, and measure the collective mutation score (i.e., the number of mutants killed by at least one test in the test suite), and collective line and branch coverage measured with the union format presented in Table 5.5.

In summary, my findings are consistent with the prior studies on effectiveness of coverage metrics [57, 96, 107, 125, 137], as I show that these metrics are generally poor indicators of the test inputs' ability in revealing system faults. However, I find that considering all lines and branches that are covered by at least one execution of the test input over multiple executions as our coverage metric results in high correlation with mutation score, and therefore can be considered a good indicator of test input and test suite quality. Finally, I show that standard deviation measurements are suitable indicators of how non-deterministic and unreliable the executions of

Table 5.4: Line and branch coverage data on all tests. Mean presents the mean coverage of all executions of the test, Mutual, Union, and Majority refer to the number of lines or branches executed by all, at least one, and the majority of test executions respectively. The last row presents the Pearson product-moment correlation of each column with the number of killed mutants in Table 5.3. * presents statistical significance.

	Line Coverage				**Branch Coverage**			
Test	**Mean**	**Mutual**	**Union**	**Majority**	**Mean**	**Mutual**	**Union**	**Majority**
0	1481.60	1468	1507	1496	638.35	613	655	643
1	1471.50	1444	1532	1481	623.65	587	705	620
2	1490.15	1476	1530	1491	653.65	618	708	640
3	1485.65	1358	1537	1512	660.65	554	724	671
4	1441.70	1347	1517	1451	606.55	535	670	601
5	1480.75	1463	1503	1491	631.50	611	650	630
6	1465.50	1453	1491	1476	617.10	600	631	617
7	1444.20	1384	1501	1480	610.00	558	656	620
8	1481.30	1460	1507	1495	630.95	604	653	633
9	1476.20	1465	1513	1489	635.60	616	660	636
10	1451.60	1441	1484	1455	604.30	585	626	603
11	1469.95	1456	1491	1481	619.15	597	633	621
12	1496.40	1459	1539	1513	668.10	599	720	684
13	1485.20	1472	1514	1490	648.60	628	673	643
14	1464.15	1452	1501	1475	612.75	599	641	612
15	1469.75	1448	1508	1479	624.80	595	670	623
Corr.	**0.29**	**0.12**	**0.66***	**0.24**	**0.33**	**0.07**	**0.59***	**0.30**

the test inputs are. I use these findings later in Section 5.4 to inform my automated test generation approach.

5.4 Automated Test Input Generation

As mentioned earlier in this chapter, the massive input space of robotic and CPSs make generating high quality test inputs extremely challenging [232]. Therefore, searching in the input space for automatically constructing meaningful test cases is one of the most important needs in the CPS testing domain [110], and as we found in our qualitative study of Section 3.1, testing *unpredictable corner cases* is one of the main challenges of testing these systems [20].

The automated test input generation approaches broadly can be divided into two categories: 1) Model-based testing, which provides techniques for automatic test case generation using models extracted from software artifacts, and 2) Search-Based Software Testing (SBST), which is a method for automated test generation based on optimization using meta-heuristics [198]. As mentioned in Section 2.1, many model-based approaches have been proposed for automated test generation of CPSs [197, 205, 277]. However, these approaches require a model of the system

Table 5.5: The collective mutation score, union line coverage, and branch coverage of 10 randomly selected test suites. The last line presents the Pearson correlation between the mutation score and the line and branch coverage. * shows significance.

Test Suite	Mutation Score	Union Line Coverage	Union Branch Coverage
TS 0	87	1536	711
TS 1	87	1539	714
TS 2	61	1517	676
TS 3	77	1540	723
TS 4	91	1543	732
TS 5	92	1540	722
TS 6	85	1543	731
TS 7	87	1539	714
TS 8	113	1542	732
TS 9	77	1541	730
Corr.		**0.68***	**0.66***

(e.g., Simulink models or finite state machines), which are difficult and error-prone to be specified [110], and may not be available for many non-safety-critical robotic and CPSs [122]. SBST approaches on the other hand, do not require models of the system, and have been shown to be effective in finding errors in CPSs (e.g., automated driving control applications [110]).

One of the families of search-based algorithms suitable for automated test generation is evolutionary algorithms (e.g., genetic algorithm) [27]. These algorithms are efficient heuristic search methods based on Darwinian evolution with powerful characteristics of robustness and flexibility to capture global solutions of complex optimization problems, and are appropriate for problems with stochastic characteristics, uncertainties or fitness with noise [99, 204]. Figure 5.5 presents an overview of a typical genetic algorithm that can be used for test case generation. First, an initial population of test inputs is provided to the algorithm. Next, this population is iteratively evolved towards higher *fitness*, that is measured by a fitness function. In each iteration, a new generation of test inputs are created by 1) selecting the parents, 2) combining the parents through *crossover* to create offsprings, and 3) mutating a number of the offsprings. This new generation is then evaluated by the fitness function, and the evolution continues until stopping criteria are met.

In this section, I present an automated test generation approach using genetic algorithm. I first describe the generation of initial population in Section 5.4.1. Next, in Section 5.4.2, I discuss the fitness functions I use for test generation. In Section 5.4.3, I present the evolution strategy and operators (i.e., crossover and mutation) used in the algorithm, and in Section 5.4.4, I evaluate the quality of generated test inputs with respect to different metrics.

5.4.1 Initial Population and Setup

in order to apply genetic algorithm to test generation as illustrated by Figure 5.5, we need to define the encoding of the population (a.k.a., the chromosomes), the total number of genera-

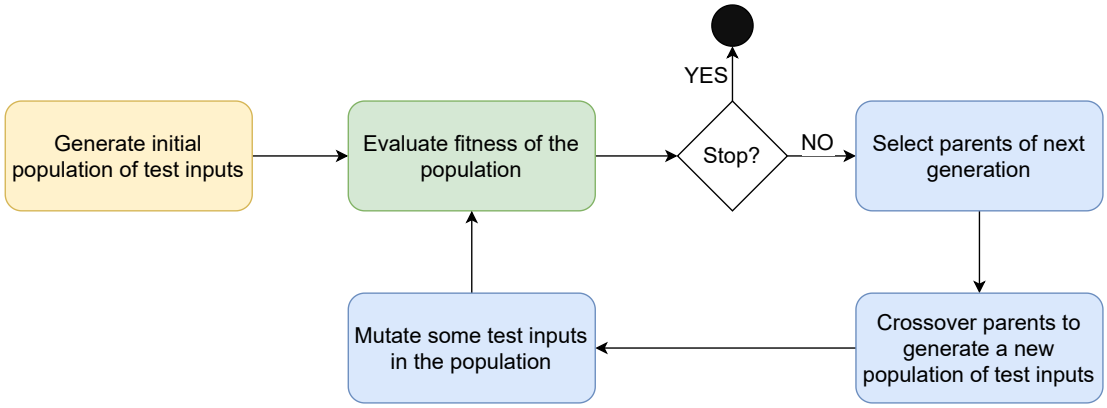


Figure 5.5: An overview of automated test input generation using genetic algorithm that includes three main components of initial population (yellow), fitness function (green), and evolution (blue).

tions (G) that the algorithm should evolve before stopping, and the number of solutions in each population (S). Additionally, we need to specify a strategy to create an initial population.

As described in Section 5.2 our test inputs are in the form of test scenarios that include both a simulation scene, and a mission to be performed by the robot. Therefore, a population in our genetic algorithm is defined as a set of S test scenarios.

As initial population for our algorithm, I randomly create test scenarios that include a single robot placed in an empty $5 \times 10(meters)$ room, with two randomly placed mission waypoints. I limit the simulation scene to a $50m^2$ room to make sure the missions can be finished in a timely manner.

5.4.2 Fitness Function

The fitness function of an evolutionary-based algorithm guides the search, and can highly affect the SBST process and its effectiveness [27]. Many prior work for example have used statement or branch coverage as fitness functions for their automated test generation algorithm [95, 177, 223, 287]. However, as I presented in Section 5.3.3, generally coverage metrics are not suitable metrics for measuring test quality in robotic systems, except the union coverage, the coverage collected over multiple runs of the test on the system, which can be a stronger metric for measuring the effectiveness of the tests. Additionally, I showed that std is a suitable measurement to determine non-determinism in test cases. However, computing both std and union coverage of a test case is expensive as they require multiple runs of the same test. Let us assume that the genetic algorithm will run for 50 generations, each consisting of 10 scenes. To compute the std and coverage metrics, we need to run each one of these generated scenes multiple (e.g. 20) times. With these numbers, our genetic algorithm needs to execute $50 \times 10 \times 20 = 10000$ executions. My prior experiments show that with 10 tests running in parallel, we can collect about 160 executions per hour, which means this algorithm needs to run for more than 62 hours. Therefore, we need to use alternative fitness functions in our genetic algorithm, and use the std and coverage metrics for evaluation.

Another fitness function that has been specifically used in the field of autonomous driving

applications is *scenario coverage*,[2] where the test generation or selection approaches focus on maximizing the diversity and effectiveness of the scenarios presented to the SUT [34, 102, 150, 202, 208, 233, 281]. For example, Xia et al. [281] create an influence factor and importance degree model for different elements of a driving scenario such as the environment (e.g., weather), positioning of the roads, and the road traffic, which they use to generate testing scenarios that are more effective in challenging the driving control system, and are diverse. In testing robotic and CPSs, we desire to expose the SUT to different, possibly all scenarios and situations that the system can be faced to ensure that the system performs as expected, in a safe manner. An important advantage of using scenario coverage metrics as fitness function for search-based test generation approaches is their low computation cost, since the fitness of a scenario is determined by its own characteristics, not the data collected over a single, or multiple test executions.

The source of scenarios is referred to the information sources that specify what factors distinguish the scenarios from each other with regards to their effectiveness. This information can be in the form of abstract knowledge from experts, standards and guidelines (i.e., knowledge-based), or can be determined from data from real world (i.e., data-based), which needs to be as comprehensive as possible [233].

I hypothesize that using the information about the testing scenario itself (e.g., the area involving the robot's path that include obstacles, and the distance between source and destination), inspired by even limited knowledge-based scenario sources, we can create scenario coverage-based fitness functions that if used with evolutionary approaches result in higher effectiveness in revealing failures in the system.

For this purpose, we need to identify characteristics of test scenarios that influence the performance of the robot (i.e., influence factors), and their level of impact on the scenarios (i.e., importance degrees) [281]. Using this information, we can propose fitness functions to guide our search algorithm towards different, possibly multiple objectives.

Scenario's Influence Factors The influence factors of a scenario are its characteristics that influence the robot's functionalities and performance [281]. By taking a closer look at the test scenarios used in Section 5.3's, experiments, I anecdotally made the following observations:

- The robot has a harder time localizing (i.e., determining the coordinates of its own location) when the scene includes fewer items. In other words, when the environment surrounding the robot is emptier, the localization error in the system increases. This observation matches with the fact that localization techniques rely on surrounding objects and signs to more accurately estimate the robot's location [154].
- If the localization module is challenged by the emptiness of the environment, the further away the mission waypoints are from the source, the higher the localization error gets.
- The local base planner and navigation function of the system get challenged when the destination of a path (i.e., mission's waypoints) is close to obstacles. For example, if the destination point is between two walls close to each other, the robot has a harder time determining the right path and local velocity commands to navigate safely without hitting obstacles.

[2]Sometimes referred to as *situation coverage* when the scenario involves dynamic agents [123, 270].

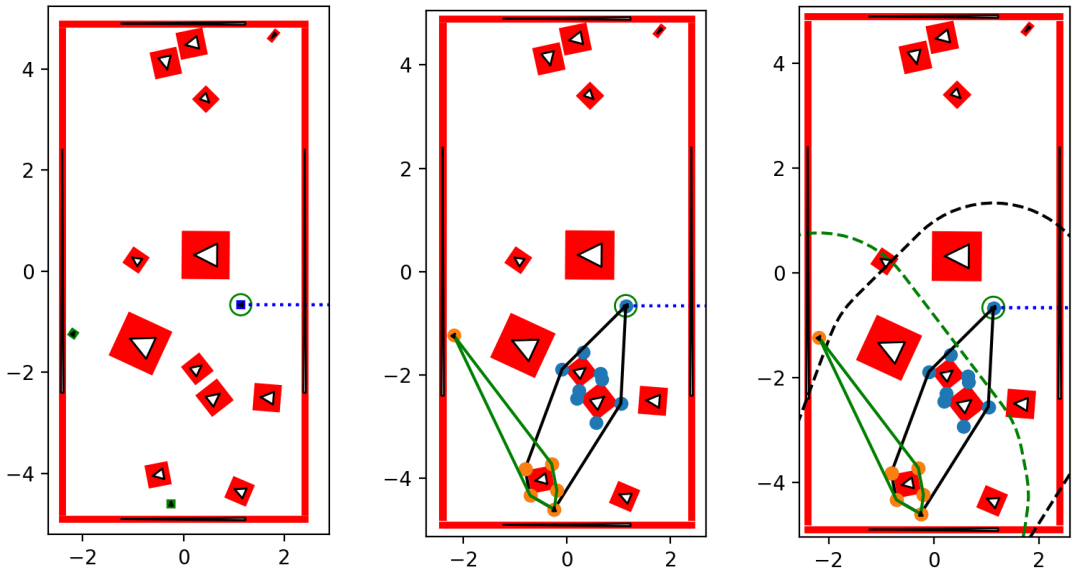


Figure 5.6: (a) The 2D plot of an exemplary test scenario where red rectangles present objects in the scene, the Blue, circled rectangle is the initial position of the robot, and the two green rectangles are the mission waypoints. (b) The path regions identified for the two navigation missions presented by polygons in solid lines. (c) The vision regions surrounding the path regions presented by dashed lines.

- The path finding of navigation function has a harder time determining a valid path between the source and the destination when the region between the two points is crowded (i.e., includes many obstacles), and the robot has to navigate through narrow corridors.

Note that I do not claim that these observations are 100% true in all scenarios as I do not consider myself an expert on the SUT. I have simply made these observations on a sample of scenarios with different features.

In order to verify these observations, and possibly use them in scenario coverage metrics, we need to measure and quantify certain aspects of the test scenario such as the emptiness of the environment surrounding the robot, or complexity of the probable path between the source and the destination. As a result, we first need to identify the region that has a high chance to include the optimal path. Note that we do not want to reimplement the path finding algorithm of the system as that defeats the purpose. Here, having an approximation of the overall region that involves the path would be sufficient.

To identify such regions, let me first refer back to the scenes generated by GzScenic described in Section 5.2. Every scene stores information related to the spatial relationship of the objects, their width and length,[3] the initial position of the robot, and the mission waypoints. Figure 5.6a presents the plot of a random scene generated by GzScenic. As we can observe in this exemplary scene, there are a number of objects in the scene that are unlikely to ever be involved in the mission such as the ones on the top of the plot. As a result, we need to automatically identify the

[3]All objects in Scenic are represented by 2D rectangles, therefore GzScenic creates a bounding box for each object in the scene.

objects that are more likely to be on the robot's path to the waypoints.

```
1: procedure IDENTIFYPATHREGION(source, destination, scene)
2:     pathRegionObjs ← ∅
3:     pathRegion ← LINE(source, destination)
4:     objects ← INTERSECTINGOBJS(pathRegion, scene)
5:     while objects ≠ ∅ do
6:         pathRegionObjs ← pathRegionObjs ∪ Objects
7:         pathRegion ← CONVEXHULL(pathRegionObjs)
8:         objects ← INTERSECTINGOBJS(pathRegion, scene) − pathRegionObjs
9:     end while
10:    return pathRegion, pathRegionObjs
11: end procedure
```

Figure 5.7: The procedure to identify the *path region*, which is a region that is likely to include the robot's path from $source$ to $destination$, and the objects in it.

For this purpose, I use the algorithm described in Figure 5.7, where a region that is likely to include the robot's path and the objects in it are identified for a pair of source and destination in our scenario. I first connect the source and destination points with a straight line. If this straight line does not intersect with any objects, then it is probable that the robot will take this straight path. However, if this line intersects with objects, the robot certainly needs to go around those objects in order to find a valid path. Therefore, my approach automatically creates a convex polygon that includes all intersecting objects, the source, and the destination. I consider the convex polygon as it provides an approximation of the *path region*, the region that has a high probability to include the path, and is cheap to compute. My approach repeats this step again by checking whether the newly created polygon intersects with any objects that are not already included in it, and if so creates another convex polygon that includes the intersecting objects as well. This is continuously done until no objects can be added to the region polygon. Figure 5.6b presents the identified path regions for the exemplary scene.

In addition to the possible path region, I am interested in identifying the area that will possibly be covered by the vision sensors of the robot. The robot's sensor are able to scan its surroundings up to some distance, which depends on the type of the sensor, and the system's configurations. TURTLEBOT3 uses Laser scanner sensor to observe its surroundings, and can scan everything in its 3.5 meters radius with some noise. Therefore, I define a larger *vision region* that surrounds the path region by two meters and the robot can more reliably scan, and consider any objects that falls in this region as *in-vision* objects. Figure 5.6c presents the vision regions with dashed line.

In addition to these interesting regions, it is important to identify the narrow corridors that the robot may be needing to pass since that can be a factor in creating more effective and diverse scenarios. For this purpose, my approach first attaches all objects in the path region that are closer to each other then the robot's width. In other words, if two objects are too close to each other, the robot cannot pass between them and therefore they can be considered as a single attached object, which I will refer to as *section*s. Figure 5.9 illustrates the procedure for identifying these sections. Figure 5.8 shows the two identified sections of a path region in the exemplary scene

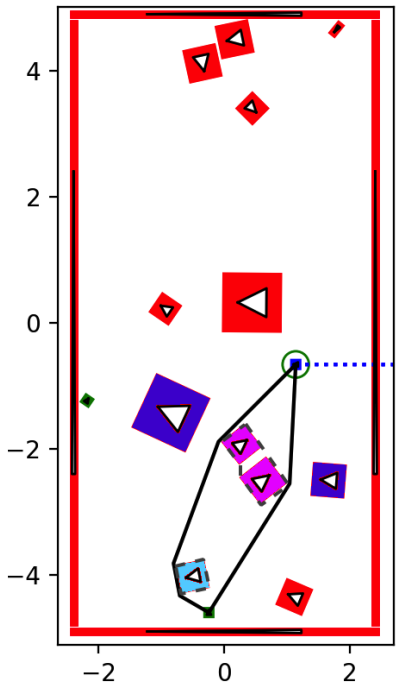


Figure 5.8: A path region (solid line polygon) in an exemplary test scenario, the two identified sections (dashed line polygons) with their objects colored pink and light blue. The two objects colored purple are close to the pink section and create narrow corridors the robot may need to pass.

presented by grey dashed line. The sections that are *splitters*, meaning that they divide the path region in two or more separate parts, give us more confidence that the robot has to move out of the path region to reach destination. In Figure 5.8 the section in pink is a splitter while the section in blue is not. Additionally, my approach automatically identifies the objects in the vision region that are close to the sections, which can indicate existence of narrow corridors between sections and other objects. In the example figure, the purple objects are in close vicinity of the pink section, and hint that corridors exist between pink section and the in-vision objects.

Table 5.6 presents the influence factors or characteristics of a scene that my approach automatically computes. Note that these factors are not necessarily independent of each other, and are separately computed for every pair of source and destination in the mission, since the difficulty of navigating from point A to B is *mostly* independent of navigating from B to C. Figure 5.10 presents the high-level algorithm of measuring these influence factors for a provided scenario. Next, I discuss how these factors can be used in scenario coverage.

Importance Degrees In order to use the influence factors described in Table 5.6 for scenario coverage and ultimately in the fitness function of evolutionary-based automated test generation, we need to identify the importance of different factors with regards to different objectives [281]. For example, the emptiness of the environment, which can be represented by a combination of TAP, TOP, TAV, and TOV factors, can be important in representing scenarios with higher localization error. However, another factor such as TAS may not be as important for localization of the robot. The importance degree of factors can be determined from experts knowledge, from data,

```
1: procedure IDENTIFYSECTIONS(pathRegion, pathRegionObjs, ROBOTWIDTH)
2:     sections ← ∅
3:     for all obj ∈ pathRegionObjs not already in sections do
4:         newSectionObjs ← {obj}
5:         repeat
6:             newSectionRegion ← CONVEXHULL(newSectionObjs)
7:             newSectionRegion ← ADDBOUNDS(newSectionRegion, ROBOTWIDTH)
8:             intObjects ← INTERSECTINGOBJS(newSectionRegion, pathRegionObjs)
9:             objects ← intObjects − newSectionObjs
10:            newSectionObjs ← intObjects
11:        until objects = ∅
12:        splitter ← ISSPLITTER(pathRegion, newSectionRegion)
13:        add SECTION(newSectionRegion, newSectionObjs, splitter) to sections
14:    end for
15:    return sections
16: end procedure
```

Figure 5.9: The procedure to identify the *sections* of objects in a path region that cannot be passed by the robot.

or a combination of both.

First, I define two main objectives for the importance degree evaluation:

1. Non-determinism: for this objective I identify the importance degree of factors in a manner that they best represent a test's maximum localization error std, and maximum closest distance to waypoints std.
2. Effectiveness: this objective targets higher mutation score (i.e., effectiveness in revealing faults)

The non-determinism objective can allow us to identify a number of sources of non-determinism in the system, and to evaluate the reliability of test scenarios to control the level of non-determinism in our tests. The effectiveness objective aims to identify importance degrees that reflect the mutation score or effectiveness of the test scenarios.

For each objective, I use my basic and limited knowledge of the system as well as the data I collected on the sample tests to learn the weights (i.e., importance degree) of different factors in the following equation:

$$Metric(x) = \sum_{f \in F} W_f \times f(x)$$

where F is the set of factors informed by the user's knowledge, W_f is the importance degree of factor f, and x is the scenario under evaluation. Note that I use a linear formula as this metric. However, the relationship between different influence factors may not necessarily be explained using a linear function. I leave investigation of other types of equations to future work.

We can either assign values to the importance factors according to experts knowledge [281], or we can use data where we attempt to maximize the correlation between $Metric(x)$ and an independent measurement related to the objective using optimization techniques. For example,

Table 5.6: The scenario influence factors that can automatically be measured for each pairs of source and destination in the mission, and for every sections identified in the path regions.

	Type	Description
Per source & dest	TAP	The total area of the path region.
	TOP	The occupied area of the path region.
	TAV	The total area of the vision region.
	TOV	The occupied area of the vision region.
	SEC	The number of sections in the path region.
	DIST	The distance between the source and destination.
	MDS	The minimum distance between the destination point and the objects surrounding it.
Per section	TAS	The total area of the section.
	SPL	Whether the section is a splitter.
	SOB	The surrounding objects to the section and their distance.
	LEN	The length of the section.

we can set values to W_fs such that $Metric(x)$ highly correlates with maximum localization error standard deviation. The limitation with this approach is the possibility of introducing bias, and overfitting to the data samples, which can be reduced by larger and more diverse set of data samples.

I use multivariate linear regression [206] to analytically find a set of importance factors (i.e., W_is) for every measurement we envision to use in our objective functions such that the correlation between $Metric(x)$ and the measurement is maximized. For instance, the maximum localization error std and maximum closest distance to the waypoints std measurements can be used for the non-determinism objective, and the mutation score and mean closest distance to the waypoints measurements can inform the effectiveness objective.

Using the influence factors, and importance degrees described earlier, I define five fitness functions for our genetic algorithm to create five different test suites that focus on either generating more non-deterministic tests, or tests with higher mutation score. I select these fitness functions to evaluate the selection of influence factors and importance degrees (whether higher $Metric(x)$ for an objective O truly result in higher levels of O among test executions), provide a means to control reliability of test inputs, and measure effectiveness of test inputs with respect to their ability in revealing faults. These five fitness functions are as follows:

1. **Nondet. LE:** $Metric(x)$ with weights focusing on standard deviation of maximum localization error.
2. **Nondet. Dist:** $Metric(x)$ with weights focusing on standard deviation of maximum closest distance to the waypoints.
3. **Nondet.:** a multi-objective [209] fitness function that considers both of the two previous metrics.
4. **Mut. Sc.:** a $Metric(x)$ with weights focusing on the mutation score.
5. **Det. Mut. Sc.:** a multi-objective fitness function that considers *Nodet. LE.* and *Nondet.*

```
1: procedure MEASUREINFLUENCEFACTORS(scenario, ROBOTWIDTH)
2:     allMeasures ← empty list
3:     scene ← all objects in scenario
4:     for all source S and destination D ∈ scenario do
5:         pathRegion, pathRegionObjs ← IDENTIFYPATHREGION(S, D, scene)
6:         visionRegion ← ADDBOUNDS(pathRegion, 2)              ▷ adding 2 meters bound
7:         visionRegionObjs ← INTERSECTINGOBJS(visionRegion, scene)
8:         sections ← IDENTIFYSECTIONS(pathRegion, pathRegionObjs, ROBOTWIDTH)
9:         add MEASURES(*) to allMeasures    ▷ * represents all previously defined variables
10:    end for
11:    return allMeasures
12: end procedure
```

Figure 5.10: The procedure to measure the influence factors for a scenario. MEASURES(*) computes the measurements for the factors of Table 5.6 based on the identified regions and sections.

Dist negatively (aiming for higher determinism), and *Mut. Sc.* positively (aiming for higher mutation score).

In Section 5.4.4, I evaluate these fitness functions using a number of different metrics.

5.4.3 Evolution

As mentioned earlier in Section 5.4 and presented by Figure 5.5, search-based, evolutionary approaches of generating test suites require the selection strategy for selecting parents for the next generation, and the various evolution operators, such as crossover and mutation operators.

First, we initialize a new population of test scenarios with a number of the best individuals of the last generation (E), known as *elitism*. Next, we use rank selection [164] to select the parents that will create offsprings for the new generation. To be able to combine two selected parents and create offsprings, we need ways to make crossovers between scenes (i.e., mix up two scenes to generate new generation), and a way to apply mutations to the scenes. As a crossover between two test scenes presented in Figure 5.11, we randomly select a separating line parallel to the x axis, and create two child scenes each inheriting all objects in the scene higher than the separating line from one parent, and lower objects from the other parent. This approach of making crossovers reduces the chance of intersecting objects in the children scenes. The mutations that can be applied to a scene include adding new objects to the scene, moving existing objects by some random vector, moving robot's initial position or the mission waypoints, and removing objects from the scene. Figure 5.12 illustrates an example mutation applied to a scene.

5.4.4 Results

In this section, I evaluate the test inputs generated by implementing a genetic algorithm with the initial population described in Section 5.4.1, the fitness functions described in Section 5.4.2,

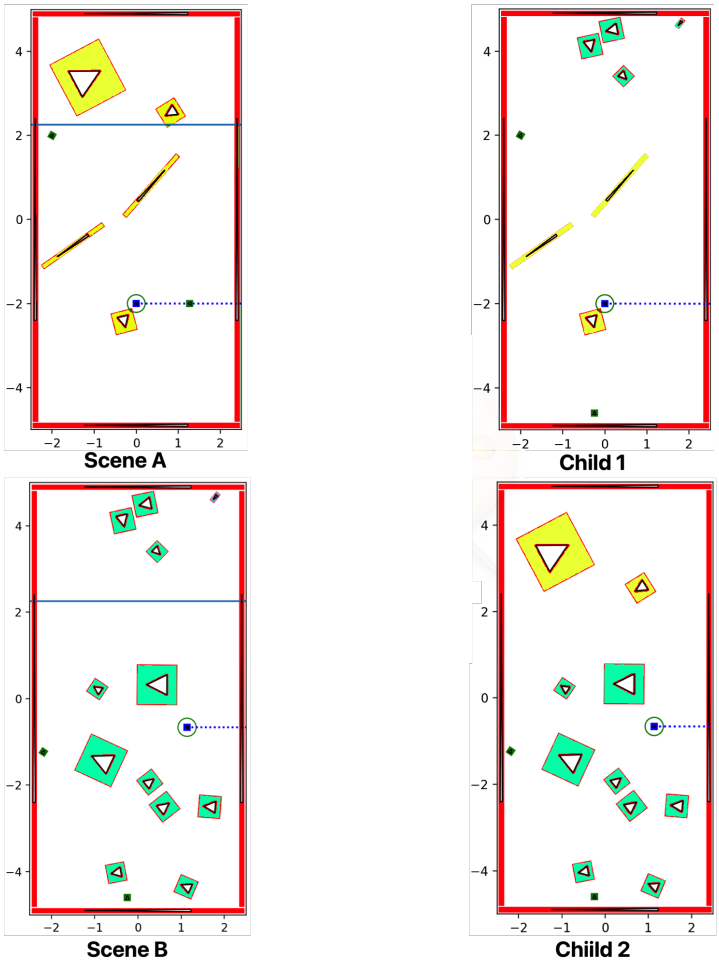


Figure 5.11: An example crossover between two scenes (left side) with respect to separating line $y = 2.2$ presented in solid blue. The resulting children (right side) inherit the objects above the separating line from one parent, and below the separating line from the other.

and evolution strategy and operators described in Section 5.4.3. For each one of the five fitness functions of Section 5.4.2, I generated a test suite of 30 tests by running the evolution for $G = 30$ generations, each population consisting of $S = 60$ solutions, and initializing a population with $E = 30$ elites from previous generation. I leave the evaluation of the impact of selecting different settings for the algorithm to future work. For each generated test suite of 30 test inputs, I ran each test 20 times, and collected the test's standard deviation on maximum localization error and maximum closest distance to the waypoints, and the union line and branch coverage. Additionally, I randomly created 30 test scenes, and computed the same measurements. Since both the genetic algorithm and random test generation involve randomness, I repeat these steps for 5 different random seeds. In total, for every fitness function and the random test suite, total number of $30 \times 5 = 150$ test inputs are generated, and the data is collected over $150 \times 20 = 3000$ test executions.

Table 5.7 presents the mean of different measurements for each test suite. As shown, the test inputs generated using *Nondet. LE* and *Nondet. Dist* fitness functions have resulted in sig-

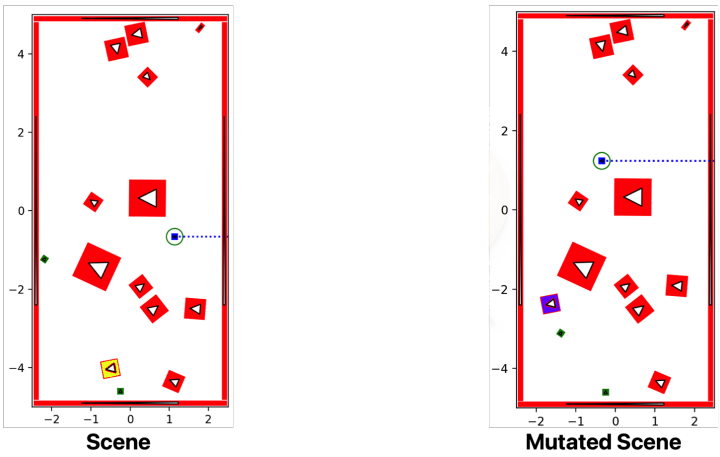


Figure 5.12: An example mutation applied to a scene (left side), resulting in the mutated scene (right side). The object in yellow has been removed from the scene, and the object in purple has been added. The initial position of the robot and one of the mission waypoints have been moved.

Table 5.7: The mean of different measurements on six test suites (1 randomly generated and 5 generated using genetic algorithm with different fitness functions) each containing 150 test cases generated with 5 different random seeds, and executed 20 times on the case study system.

	Mean of				
Test Suite	**Max LE Std**	**Max Dist Std**	**Sum Std**	**Line Cov.**	**Branch Cov.**
Random	0.07	0.29	0.35	1503.31	660.16
Nondet. LE	**0.14**	0.34	**0.49**	1506.18	**674.17**
Nondet. Dist.	0.05	**0.78**	**0.83**	**1514.61**	**684.95**
Nondet.	0.08	0.34	0.41	1509.62	**677.82**
Mut. Sc.	**0.14**	0.37	**0.51**	**1513.79**	**681.68**
Det. Mut. Sc.	0.10	**0.48**	**0.57**	1505.20	**668.98**

nificantly higher standard deviation in their maximum localization error and maximum closest distance to the waypoints respectively, compared to the randomly-generated test inputs. These results suggest that the influence factors and importance degrees calculated for each one of these objective can truly represent the objective. By examining the test scenarios generated for these two fitness functions, and the importance degrees computed for different influence factors of the scenario, I made the observation that the non-determinism with respect to localization error increases as the scene is emptier. On contrary, the non-determinism in performing the mission (i.e., maximum closest distance to waypoints) is increased as the scene includes higher number of obstacles. As these two objectives are in contrast with each other, using a multi-objective approach to maximize both objectives is ineffective as presented by the *Nondet.* test suite. Overall, these results suggest that the non-determinism in test executions can be increased or decreased, once we identify their sources, and have a way of manipulating these sources. I further discuss

this idea and its possible impact on the quality of robotic and CPSs in future work.

Table 5.7 also presents the performance of test suites generated using fitness functions that aim to maximize the mutation score (i.e., *Mut. Sc.* and *Det. Mut. Sc.*). Since using mutation score for the evaluation of the generated tests is extremely expensive, even infeasible, I use union coverage as a proxy to measure fault revealing effectiveness of tests as I showed earlier that it has a high and strong correlation with the mutation score. Table 5.7 shows that the sole metric of maximizing mutation score significantly improves the union line coverage and union branch coverage compared to the random tests, but it also results in very high standard deviation of maximum localization error. A multi-objective approach to generate deterministic and high coverage tests is effective with respect to higher union branch coverage, and it does not introduce high levels of non-determinism with respect to localization error. However, it still introduces a high level of non-determinism in the maximum closes distance to the waypoints, which can arise from the fact that the robot struggles to reach the mission waypoints in the more challenging scenarios.

Overall, these results suggest that low cost, scenario coverage-based metrics, inspired by limited knowledge about the SUT, can be effective in generating test scenarios for different objectives, and encourage further studies in this area to propose more advanced methods of determining the influence factors, the importance degrees, and eventually the fitness functions. In Section 5.6, I discuss a number of ideas that can possibly improve the approach presented in this chapter.

5.5 Limitations and Threats

Limitations The automated test generation approach presented in Section 5.4 can only be applied to systems where the simulation scenarios can be abstracted and encoded by different influence factors that require knowledge about the SUT.

To control the scope of the work presented in this chapter I limited the experiments to scenarios with a single robot agent, in a static environment, under the default configuration setting. I leave efficient exploration of the configuration space for testing to future work.

Threats to Validity Throughout this chapter, I performed studies on a case study robotic system, TURTLEBOT3, to test ROS's navigation planner packages. Even though these packages are used by thousands of users and systems, I cannot make any claims about generalizability of the results in this work. Therefore, the selected case study system not being a suitable representative of robotic systems is a threat to the external validity of my studies.

The size of our data, and the methodology of collecting data in Section 5.3 and 5.4 can be a source of threats to internal validity. For example, in Section 5.3, I used 16 manually and randomly specified test scenarios, which may not be true representative of diverse set of scenarios for the system, and bias the findings of the study. Additionally, random operations can introduce a threat to validity of the results, which I attempt to mitigate by using multiple random seeds for different experiments.

The influence factors used in Section 5.3 are defined based on my limited knowledge of the SUT. Since I do not consider myself as an expert on the SUT, this knowledge is a threat to the

internal validity.

In the experiment of Section 5.4, we used union coverage as a proxy for mutation score. Since union coverage is not 100% correlated with mutation score, we cannot be certain that test inputs with higher union coverage are better at revealing faults, which can be a threat to external validity of the results.

5.6 Summary and Future Work

To summarize, in this chapter, I focused on automatically generating high quality test inputs that can be executed in simulation for a case study robotic system. I first presented GzScenic, a tool I created that allows for easy and automated scenario construction. Second, I studied the quality of different test inputs with respect to their reliability and ability to reveal faults. I showed non-deterministic executions can challenge evaluation of test inputs, and that coverage-based quality metrics are generally poor indicators of the test's effectiveness, unless we consider union coverage of multiple executions of the same test.

Finally, to automatically generate high-quality test inputs, I proposed an evolutionary-based test generation approach with a fitness function that is based on *scenario coverage*, where the test generation or selection approach focuses on maximizing the diversity and effectiveness of the scenarios presented to the system under test. I showed that this approach is effective in generating high-quality tests.

Even though the experiments in this chapter have only been conducted on a single system, the methodology and approach used in this work can be applied to other, more complex systems in the future. Additionally, we can propose more advanced and accurate models for identifying the influence factors and the importance degrees discussed in Section 5.4.2. For example, instead of using a linear function too describe relationship between different influence factors and their corresponding importance degrees, we can use neural networks that are capable of detecting more complex equations.

In Section 5.3.2, I briefly discussed non-determinism in test executions, its possible sources, and how it impacts testing. The results of this study motivates further investigation of this matter in robotic systems in the future. For instance, future studies can investigate different sources of non-determinism impacting the robot's performance in simulation, and propose techniques to control or adjust non-determinism in simulation. On top of that, automated tools can monitor the SUT's executions in simulation and perform an analysis to identify the components or subsystems with highest rate of non-determinism and uncertainty. This information can help developers to make their system more deterministic, or use artificial data or mocking to replace the non-deterministic components for the purpose of achieving repeatable, reproducible tests.

In this work, I presented a set of influence factors that provide and abstraction of the scenario characteristics that impact test executions. Future work can build on top of the ideas presented in this chapter, and present more accurate and representative influence factors. For example, taking ideas from prior work on measuring shape complexity [54, 238], we can provide more accurate measures of how complex the simulation scene is regarding a navigation mission. Additionally, we can take into account the complexity of different obstacle objects, as they may be differently challenging fr the robot to get around.

Finally, in future we can use the test generation approach proposed in this chapter, and the oracles of Chapter 4 on a system to achieve and evaluate a large-scale, automated testing pipeline. We can further expand the approaches presented in this thesis to move towards better automated quality assurance of robotic and cyberphysical systems, by taking advantage of automated fault localization and automated program repair. However, more studies need to be performed to investigate effectiveness of these automated techniques on robotic and CPSs.

Chapter 6

Conclusions and Final Remarks

In summary, this thesis covers a number of empirical and non-empirical studies with the purpose of improving automated testing of robotic and cyberphysical systems. The empirical studies identify the state and challenges of testing CPSs, while non-empirical studies propose an automated testing pipeline that improves the quality assurance of these systems in simulated environments.

My work encourages more research and studies to be conducted in the field of automated quality assurance of robotic and cyberphysical systems by identifying the specific challenges of testing these systems in simulation and illustrating the advantages of automated simulation-based testing. In addition, the approaches I introduce in this thesis, which includes automated oracle inference and automated test generation, improve current state of automated testing, and enable future studies on improving different pieces of the proposed techniques.

The advancements in the field of robotics are taking place at an unimaginable pace. In 2019, more than 80 companies in the United States alone were reported to be developing autonomous driving vehicles, and testing over 1,400 of their vehicles on the roads [90], and possibly more companies are working on autonomous driving technology but have not yet reached the point of testing their systems on public roads. Even though most of these products are not yet mature enough to be fully-deployed in the field, and being used by the users, they are not far from it, and may actually reach that maturity in the next few years. As a result, now is the time to develop and establish methods that ensure the quality of these systems to prevent catastrophic incidents.

I believe fully-automated, large-scale testing is the best solution to the problem of quality assurance of robotic and CPSs, especially with respect to the software. Without a fully-automated approach, our testing will be limited by its scalability, which is extremely important for systems that interact with the physical world. However, we have a long way to achieve fully-automated testing (possibly in the format of continuous integration) of robotic and CPSs in practice, which is mainly caused by the special features of these systems such as interacting with the physical world, the non-determinism, and integration of many subsystems. The software engineering community, along with the robotics researchers and practitioners, need to focus on addressing the challenges of deploying fully-automated, large-scale testing for these systems.

Some challenges that are discussed in this dissertation require more advanced engineering that is capable of handling highly complex systems, with features like non-determinism. For example, better tooling and guidelines can reduce the cost of building and maintaining an end-to-

end test harnesses for robotic systems. Other challenges require novel techniques and approaches that can address specific requirements of these systems. For example, both the oracle problem and effective test input generation are challenges that cannot simply be addressed by more engineering effort. Even though addressing the engineering challenges may not be considered novel and important by the research community, in my opinion, they are equally, if not more important than the research challenges in achieving higher rate of automated testing being deployed on these systems. Therefore, future studies should focus on both of these groups of challenges, and propose more advanced techniques and tools.

In addition to the quality assurance of the software (cyber) component of these systems, we need to consider the hardware (physical) component of these systems as well. Automated, hardware-in-the-loop testing (briefly mentioned in Chapter 3) is a promising approach that allows us to apply quality assurance to the integration of software and hardware components [92], which is an extremely important factor for these systems. However, we have very limited knowledge about the effectiveness and popularity of HITL testing, and the challenges that prevent higher deployment of such techniques in practice. In fact, our qualitative studies with robotics practitioners showed little evidence of HITL testing being used in practice. Future studies on the quality assurance of robotic and CPSs that involve the hardware can significantly contribute to the overall quality of these systems.

This thesis contributes to the automated testing and quality assurance of robotic and cyberphysical systems in the following ways (restated from Section 1.2):

1. It identifies the challenges of automated testing for robotics systems and discovers the practices currently being used in the field of robotics.
2. It shows that simulation-based testing can be an effective approach in identifying faults in these systems.
3. It identifies the challenges of using simulators for the purpose of (automated) testing, and the most prominent issues with currently available simulators.
4. It presents a black-box approach to automatically infer oracles for these systems based on observed executions of the robot in the simulated environment.
5. It investigates the severity of non-determinism among test executions in simulation, and offers insight on the performance of coverage-based quality metrics as indicator of test inputs fault-revealing effectiveness.
6. It presents an evolutionary-based automated test generation approach using scenario coverage as fitness function.

Overall, by identifying the challenges in testing robotic and cyberphysical systems, and proposing approaches that address a subset of those challenges, this thesis takes us one step closer towards large-scale, automated testing of these systems, which eventually results in higher quality systems.

Bibliography

[1] About ArduPilot. https://ardupilot.org/index.php/about. Accessed: 2021-04-15. 4.1.1

[2] Boeing issues 737 max fleet bulletin on AoA warning after lion air crash. https://theaircurrent.com/aviation-safety/boeing-nearing-737-max-fleet-bulletin-on-aoa-warning-after-lion-air-crash/, . Accessed: 2021-04-15. 1

[3] Assumptions used in the safety assessment process and the effects of multiple alerts and indications on pilot performance. https://www.ntsb.gov/investigations/AccidentReports/Reports/ASR1901.pdf, . Accessed: 2021-04-15. 1

[4] Simulation Becomes Increasingly Important For Self-Driving Cars. https://www.forbes.com/sites/davidsilver/2018/11/01/simulation-becomes-increasingly-important-for-self-driving-cars/#56b1fa045583. Accessed: 2021-04-15. 3.1.2

[5] Husky. https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/. Accessed: 2021-04-15. 5.1

[6] IEC standards. https://webstore.iec.ch/publication/30410. Accessed: 2021-04-15. 3.1.3

[7] ISO standards. https://www.iso.org/standards.html. Accessed: 2021-04-15. 3.1.3

[8] NVIDIA Driver Constellation. https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/. Accessed: 2021-04-15. 3.1.2

[9] PR2. http://www.willowgarage.com/pages/pr2/overview. Accessed: 2021-04-15. 5.1

[10] TurtleBot. https://www.turtlebot.com. Accessed: 2021-04-15. 2.2, 5.1

[11] Ul standards. https://ulstandards.ul.com. Accessed: 2021-04-15. 3.1.3

[12] A Waymo engineer told us why a virtual-world simulation is crucial to the future of self-driving cars. https://www.businessinsider.com/waymo-engineer-explains-why-testing-self-driving-cars-virtually-is-critical-2018-8. Accessed: 2021-04-15. 3.1.2

[13] Schiaparelli landing investigation makes progress. https://www.esa.int/Science_Exploration/Human_and_Robotic_Exploration/

Exploration/ExoMars/Schiaparelli_landing_investigation_completed, 2016. Accessed 2021-04-15. 1

[14] Alireza Abbaspour, Kang K Yen, Shirin Noei, and Arman Sargolzaei. Detection of fault data injection attack on UAV using adaptive neural network. *Procedia computer science*, 95:193–200, 2016. 2.1, 4.3.1

[15] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In *International Conference on Testing Software and Systems*, pages 194–207. Springer, 2015. 1

[16] Mahmoud Abdelgawad, Sterling McLeod, Anneliese Andrews, and Jing Xiao. Model-based testing of a real-time adaptive motion planning system. *Advanced Robotics*, 31(22): 1159–1176, 2017. 2.1

[17] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2(4):433–459, 2010. 4.2

[18] Mikhail Afanasov, Aleksandr Iavorskii, and Luca Mottola. Programming support for time-sensitive adaptation in cyberphysical systems. *ACM SIGBED Review*, 14(4):27–32, 2018. 2.2, 4.5

[19] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation*, ICST '13, pages 352–361, 2013. 1, 2.1, 4

[20] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '20. IEEE, 2020. 1, 1.2, 3, 3.1, 3.3, 3.3.1, 3.3.3, 5, 5.4

[21] Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher Steven Timperley. Simulation for robotics test automation: Developer perspectives. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 1.2, 3, 3.3, 5.2, 5.3.2

[22] Afsoon Afzal, Claire Le Goues, and Christopher S. Timperley. GzScenic: Automatic scene generation for gazebo simulator. arXiv preprint arXiv:2104.08625, 2021. 1.2, 5.2

[23] Saeed Aghabozorgi and Teh Ying Wah. Clustering of large time series datasets. *Intelligent Data Analysis*, 18(5):793–817, 2014. 4.3, 4.3

[24] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. Time-series clustering–a decade review. *Information Systems*, 53:16–38, 2015. 4.3

[25] Hamdi A Ahmed and Jong-Wook Jang. Cloud based simultaneous localization and mapping with turtlebot3. In *Proceedings of the Korean Institute of Information and Commucation Sciences Conference*, pages 241–243. The Korea Institute of Information and Commucation Engineering, 2018. 2.2, 5.1

[26] Adam Alami, Yvonne Dittrich, and Andrzej Wasowski. Influencers of quality assurance in an open source community. In *International Workshop on Cooperative and Human*

Aspects of Software Engineering, CHASE'18, pages 61–68. IEEE, 2018. 2.1, 3.4

[27] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Transactions on Software Engineering*, 36(6):742–762, 2009. 2.1, 5.4, 5.4.2

[28] Shaukat Ali, Hong Lu, Shuai Wang, Tao Yue, and Man Zhang. Uncertainty-wise testing of cyber-physical systems. In *Advances in Computers*, volume 107, pages 23–94. Elsevier, 2017. 2.1, 3.1.2

[29] Maryam Raiyat Aliabadi, Amita Ajith Kamath, Julien Gascon-Samson, and Karthik Pattabiraman. ARTINALI: dynamic invariant detection for cyber-physical system security. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, pages 349–361, 2017. 1, 2.1, 4.4.2, 4.5

[30] Cesare Alippi, Stavros Ntalampiras, and Manuel Roveri. Model-free fault detection and isolation in large-scale cyber-physical systems. *Transactions on Emerging Topics in Computational Intelligence*, 1(1):61–71, 2016. 2.1, 4.3.1

[31] Hussein Almulla and Gregory Gay. Learning how to search: generating exception-triggering tests through adaptive fitness function selection. In *International Conference on Software Testing, Validation and Verification*, ICST '20, pages 63–73. IEEE, 2020. 2.1

[32] Robin Amsters and Peter Slaets. Turtlebot 3 as a robotics education platform. In *International Conference on Robotics in Education*, RiE '19, pages 170–181. Springer, 2019. 2.2, 5.1

[33] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *International Conference on Software Engineering*, ICSE '14, pages 573–583, 2014. 5.3

[34] Paolo Arcaini, Xiao-Yi Zhang, and Fuyuki Ishikawa. Targeting patterns of driving characteristics in testing autonomous driving systems. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 2.1, 5.4.2

[35] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34, 2019. 2.1

[36] J. C. M. Baeten. A brief history of process algebra. *Theoretical. Computer Science*, 335 (2-3):131–146, May 2005. ISSN 0304-3975. 2

[37] Anthony Bagnall and Gareth Janacek. Clustering time series with clipped data. *Machine Learning*, 58(2-3):151–178, 2005. 4.2

[38] Gerrit Bagschik, Till Menzel, and Markus Maurer. Ontology based scene creation for the development of automated vehicles. In *Intelligent Vehicles Symposium*, IV'18, pages 1813–1820. IEEE, 2018. 2.1

[39] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2012. 2.1

[40] Radu Banabic. Techniques for identifying elusive corner-case bugs in systems software. Technical report, EPFL, 2015. 3.1.2

[41] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 41(5): 507–525, 2015. 1, 2.1, 3.1.2, 4

[42] Mark Belko. Airport using robots, UV light to combat COVID-19. *Pittsburgh Post-Gazette*, 2020. Accessed: 2021-04-15. 1

[43] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*, ICSE '18, pages 433–444. IEEE, 2018. 2.1, 5.3, 5.3.2

[44] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-27755-2. 2

[45] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, 1994. 4.2, 4.2

[46] David Berreby. The pandemic has been good for one kind of worker: robots. *National Geographic*, 2020. Accessed: 2021-04-15. 1

[47] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, FOSE'07, pages 85–103. IEEE Computer Society, 2007. 1

[48] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *International Conference on Software Engineering*, ICSE '14, pages 468–479, 2014. 1, 2.1

[49] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Queue*, 14(2):91–110, 2016. 2.1

[50] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *Transactions on Computers*, 100(6):592–597, 1972. 1, 2.1

[51] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, 2009. 3.2.3

[52] Joshua Bloch. How to design a good API and why it matters. In *Companion to Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 506–507, 2006. 3.3.3

[53] Christopher Boelmann, Lorenz Schwittmann, Marian Waltereit, Matthäus Wander, and Torben Weis. Application-level determinism in distributed systems. In *International Conference on Parallel and Distributed Systems*, ICPADS '16, pages 989–998. IEEE, 2016. 1, 2.1, 3.3.3, 5, 5.3.2

[54] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Alexander Braun. Measuring the complexity of polygonal objects. In *ACM-GIS*, volume 109. Citeseer, 1995. 5.6

[55] Frederick P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987. 3.3.2

[56] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982. 5.3

[57] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *International Workshop on Advances in Model-Based Testing*, A-MOST '05, page 1–7, 2005. 1, 2.1, 5.3, 5.3.1, 5.3.3, 5.3.3

[58] Matteo Camilli, Angelo Gargantini, Patrizia Scandurra, and Catia Trubiani. Uncertainty-aware exploration in model-based testing. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 2.1

[59] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. An industrial survey on contemporary aspects of software testing. In *International Conference on Software Testing, Verification and Validation*, ICST'10, pages 393–401. IEEE, 2010. 1, 2.1

[60] Kathy Charmaz. *Constructing Grounded Theory*. Sage, 2014. 3.3.1, 3.3.1

[61] Kathy Charmaz and Linda Liska Belgrave. Grounded theory. *The Blackwell encyclopedia of sociology*, 2007. 1.2

[62] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *Computing Surveys*, 33(3):273–321, 2001. 3

[63] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4, 2018. 3.1.3

[64] Yunliang Chen, Lizhe Wang, Fangyuan Li, Bo Du, Kim-Kwang Raymond Choo, Houcine Hassan, and Wenjian Qin. Air quality data clustering using epls method. *Information Fusion*, 36:225–232, 2017. 4.2

[65] Yuqi Chen, Christopher M. Poskitt, and Jun Sun. Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system. In *Symposium on Security and Privacy*, S&P '18, pages 648–660, 2018. 2.1, 3.1.3, 3.1.3, 4.4.2, 4.5

[66] Long Cheng, Ke Tian, and Danfeng Daphne Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Computer Security Applications Conference*, pages 315–326, 2017. 2.1

[67] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. Detecting attacks against robotic vehicles: A control invariant approach. In *Conference on Computer and Communications Security*, CCS '18, pages 801–816, 2018. 2.1

[68] Marcus Ciolkowski, Oliver Laitenberger, Sira Vegas, and Stefan Biffl. *Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering*, pages 104–128. Springer Berlin Heidelberg, 2003. 3.3.1, 3.3.1

[69] Mariana Sátiro Coelho. *Patterns in financial markets: Dynamic time warping*. PhD thesis, NSBE-UNL, 2012. 4.2

[70] Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press, 2014. 1

[71] Daniel Cook, Andrew Vardy, and Ron Lewis. A survey of AUV and robot simulators for multi-vehicle operations. In *Autonomous Underwater Vehicles*, AUV '14, pages 1–8. IEEE, 2014. 1

[72] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi. Fault triggers in open-source software: An experience report. In *International Symposium on Software Reliability Engineering*, ISSRE '13, pages 178–187, 2013. 3.2, 3.2.3

[73] Jeff Craighead, Robin Murphy, Jenny Burke, and Brian Goldiez. A survey of commercial & open source unmanned vehicle simulators. In *Proceedings of International Conference on Robotics and Automation*, ICRA'07, pages 852–857. IEEE, 2007. 1

[74] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering*, ICSE '08, pages 281–290. ACM, 2008. 1, 2.1

[75] Mirella Santos Pessoa de Melo, José Gomes da Silva Neto, Pedro Jorge Lima da Silva, João Marcelo Xavier Natario Teixeira, and Veronica Teichrieb. Analysis and comparison of robotics 3D simulators. In *Symposium on Virtual and Augmented Reality*, SVR '19, pages 242–251, 2019. 2.2, 3.3

[76] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. 1, 2.1, 5, 5.3, 5.3.3

[77] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*, pages 172–189. Springer, 2017. 1, 3.1.3

[78] Ankush Desai, Shaz Qadeer, and Sanjit A Seshia. Programming safe robotics systems: Challenges and advances. In *International Symposium on Leveraging Applications of Formal Methods*, pages 103–119. Springer, 2018. 1, 3.1.3

[79] I Dhiah el Diehn, Sahel Alouneh, Roman Obermaisser, et al. Incremental, distributed, and concurrent service coordination for reliable and deterministic systems-of-systems. *IEEE Systems Journal*, 2020. 3.3.3

[80] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 1 October 2005. 3.2

[81] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Formal requirement debugging for testing and verification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 17(2):34, 2018. 3.1.3

[82] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Conference on Robot Learning*, CoRL, pages 1–16, 2017. 2.2

[83] Pengfei Duan, Ying Zhou, Xufang Gong, and Bixin Li. A systematic mapping study on the verification of cyber-physical systems. *IEEE Access*, 6:59043–59064, 2018. 2.1, 2.1

[84] EdgeCase. Ul 4600: The first comprehensive safety standard for autonomous products. https://edge-case-research.com/ul4600/, 2019. Accessed: 2021-04-15. 3.1.2, 3.1.3

[85] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, 2001. 4.3.1, 4.5

[86] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. In *International Conference on Product Focused Software Process Improvement*, pages 3–16. Springer, 2010. 1, 2.1

[87] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *International Conference on Robotics and Automation*, ICRA'15, pages 4397–4404. IEEE, 2015. 2.2, 3.1.3

[88] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007. 1, 2.1, 4.5

[89] Lukas Esterle and Radu Grosu. Cyber-physical systems: challenge of the 21st century. *e & i Elektrotechnik und Informationstechnik*, 133(7):299–303, 2016. 1, 2.1, 3.1.2

[90] Darrell Etherington. Over 1,400 self-driving vehicles are now in testing by 80+ companies across the us. https://techcrunch.com/2019/06/11/over-1400-self-driving-vehicles-are-now-in-testing-by-80-companies-across-the-u-s/, 2019. Accessed: 2021-04-15. 6

[91] Marie Farrell, Matt Luckcuck, and Michael Fisher. Robotics and integrated formal methods: necessity meets opportunity. In *International Conference on Integrated Formal Methods*, pages 161–171. Springer, 2018. 1, 3.1.3

[92] Hosam K. Fathy, Zoran S. Filipi, Jonathan Hagena, and Jeffrey L. Stein. Review of hardware-in-the-loop simulation and its prospects in the automotive area. In Kevin Schum and Alex F. Sisti, editors, *Modeling and Simulation for Military Applications*, volume 6228, pages 117–136. International Society for Optics and Photonics, SPIE, 2006. 3.1.3, 6

[93] Donald Firesmith. The challenges of testing in a non-deterministic world. https://insights.sei.cmu.edu/blog/the-challenges-of-testing-in-a-non-deterministic-world/, 2017. Accessed: 2021-04-15. 1, 2.1, 5, 5.3, 5.3.2

[94] Martin Fowler. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html, 2011. Accessed: 2021-04-15. 2.1, 5.3, 5.3.2

[95] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Transactions on Software Engineering*, 39(2):276–291, 2012. 2.1, 2.1, 5.3, 5.3.3, 5.4.2

[96] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study.

ACM Transactions on Software Engineering Methodology, 24(4), September 2015. 1, 2.1, 5.3, 5.3.3

[97] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Conference on Programming Language Design and Implementation*, PLDI '19, pages 63–78, 2019. 1, 5, 3.3.2, 5, 5.2, 5.2, 5.3.1

[98] Ada Wai-Chee Fu, Eamonn Keogh, Leo Yung Lau, Chotirat Ann Ratanamahatana, and Raymond Chi-Wing Wong. Scaling and time warping in time series querying. *International Journal on Very Large Data Bases*, 17(4):899–921, 2008. 4.3.2

[99] B. Galvan, Greiner D., Périaux J., M. Sefrioui, and G. Winter. Parallel evolutionary computation for solving complex cfd optimization problems : A review and some nozzle applications. In K. Matsuno, A. Ecer, N. Satofuka, J. Periaux, and P. Fox, editors, *Parallel Computational Fluid Dynamics 2002*, pages 573–604. North-Holland, Amsterdam, 2003. ISBN 978-0-444-50680-1. 5.4

[100] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'19, pages 257–267, 2019. 2.1, 3.1.3

[101] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *International Symposium on Software Testing and Analysis*, ISSTA '19, pages 318–328, 2019. 3.1.3

[102] Feng Gao, Jianli Duan, Zaidao Han, and Yingdong He. Automatic virtual test technology for intelligent driving systems considering both coverage and efficiency. *IEEE Transactions on Vehicular Technology*, 2020. 1, 2.1, 5, 5.4.2

[103] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *International Conference on Software Engineering*, ICSE '20, pages 385–396, 2020. 2.1

[104] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: A perspective from the service robotics domain. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, pages 593–604, 2020. 2.1

[105] Vahid Garousi and Mika V. Mäntylä. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, 76:92–117, 2016. 3.3.2, 3.3.3

[106] Vahid Garousi and Junji Zhi. A survey of software testing practices in canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013. 1, 2.1

[107] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015. 1, 2.1, 5.3, 5.3.3

[108] Amin Ghafouri, Aron Laszka, Abhishek Dubey, and Xenofon Koutsoukos. Optimal detec-

tion of faulty traffic sensors used in route planning. In *International Workshop on Science of Smart City Operations and Platforms Engineering*, pages 1–6, 2017. 2.1

[109] Amin Ghafouri, Yevgeniy Vorobeychik, and Xenofon Koutsoukos. Adversarial regression for detecting attacks in cyber-physical systems. In *International Conference on Artificial Intelligence*, ICOAI '18, pages 3769–3775, 2018. 2.1

[110] Christoph Gladisch, Thomas Heinz, Christian Heinzemann, Jens Oehlerking, Anne von Vietinghoff, and Tim Pfitzer. Experience paper: Search-based testing in automated driving control applications. In *International Conference on Automated Software Engineering*, ASE'19, pages 26–37. IEEE, 2019. 1, 2.1, 2.1, 4.1.1, 5, 5.4

[111] Carlos A González, Mojtaba Varmazyar, Shiva Nejati, Lionel C Briand, and Yago Isasi. Enabling model testing of cyber-physical systems. In *International Conference on Model Driven Engineering Languages and Systems*, MODELS'18, pages 176–186. ACM, 2018. 3.1.3

[112] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *International Conference on Software Engineering*, ICSE '18, pages 1149–1159, 2018. 1, 2.1

[113] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12, 1986. 3.1.2

[114] M. Grottke, A. P. Nikora, and K. S. Trivedi. An empirical investigation of fault types in space mission system software. In *Dependable Systems Networks*, DSN '10, pages 447–456, 2010. 3.2, 3.2.3

[115] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of flaky tests in python. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 2.1, 5.3, 5.3.2

[116] Jie Gu and Xiaomin Jin. A simple approximation for dynamic time warping search in large time series database. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 841–848. Springer, 2006. 4.2

[117] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011. 2

[118] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is this a bug or an obsolete test? In *European Conference on Object-Oriented Programming*, pages 602–628. Springer, 2013. 2.1, 5.3, 5.3.2

[119] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel C Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *International Conference on Software Testing, Validation and Verification*, ICST '20, pages 85–95. IEEE, 2020. 2.1

[120] Yoshiyuki Harada, Yoriyuki Yamagata, Osamu Mizuno, and Eun-Hye Choi. Log-based anomaly detection of cps using a statistical method. In *International Workshop on Empirical Software Engineering in Practice*, IWESEP '17, pages 1–6, 2017. 2.1

[121] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and

Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, pages 851–862, 2020. 2.1

[122] Florian Hauer, Alexander Pretschner, Maximilian Schmitt, and Markus Groetsch. Industrial evaluation of search-based test generation techniques for control systems. In *International Symposium on Software Reliability Engineering Workshops*, ISSREW'17, pages 5–8. IEEE, 2017. 1, 2.1, 4.1.1, 5, 5.4

[123] Heather Hawkins and Rob Alexander. Situation coverage testing for a simulated autonomous car - an initial case study. *CoRR*, abs/1911.06501, 2019. URL `http://arxiv.org/abs/1911.06501`. 2.1, 5.3, 2

[124] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. A system identification based oracle for control-cps software fault localization. In *International Conference on Software Engineering*, ICSE '19, pages 116–127, 2019. 2.1, 2.2, 3.1.2, 3.1.3, 3.1.3, 4.3.1, 4.4, 4.4.1, 4.4.2, 4.4.2, 4.5

[125] H. Hemmati. How effective are code coverage criteria? In *International Conference on Software Quality, Reliability and Security*, pages 151–156, 2015. 1, 2.1, 5.3, 5.3.3

[126] K. Henningsson and C. Wohlin. Assuring fault classification agreement - an empirical evaluation. In *International Symposium on Empirical Software Engineering*, ISESE '04, pages 95–104, 2004. 3.2

[127] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *International Conference on Software Engineering*, volume 2 of *ICSE '15*, pages 39–48. IEEE, 2015. 1, 2.1, 5, 5.3.2

[128] Robert M Hierons. Avoiding coincidental correctness in boundary value analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):227–241, 2006. 5.3

[129] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *International Conference on Automated Software Engineering*, ASE '16, pages 426–437, 2016. 3.3.2

[130] Michael W Hofbaur and Brian C Williams. Mode estimation of probabilistic hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 253–266, 2002. 2.1

[131] J. R. Horgan, S. London, and M. R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994. 2.1, 5.3

[132] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417, 1933. 4.2

[133] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. Robustness testing of autonomy software. In *International Conference on Software Engineering - Software Engineering in Practice*, ICSE-SEIP '18, pages 276–285, 2018. 1, 2.1, 2.1, 3.1.2, 3.1.3

[134] Ignition Robotics. Ignition Gazebo: A Robotic Simulator. `https://`

ignitionrobotics.org/libs/gazebo. Accessed: 2021-04-15. 2.2

[135] Félix Ingrand. Recent trends in formal validation and verification of autonomous robots software. In *International Conference on Robotic Computing*, IRC'19, pages 321–328. IEEE, 2019. 1, 3.1.3, 3.1.3

[136] Jun Inoue, Yoriyuki Yamagata, Yuqi Chen, Christopher M Poskitt, and Jun Sun. Anomaly detection for a water treatment system using unsupervised machine learning. In *International Conference on Data Mining Workshops*, ICDMW '17, pages 1058–1065, 2017. 2.1, 3.1.3, 4.3.1, 4.5

[137] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*, ICSE '14, page 435–445, 2014. 1, 2.1, 5.3, 5.3.1, 5.3.3, 5.3.3

[138] Max Planck Institute. Robotics and cyber-physical systems. https://www.cis.mpg.de/robotics/. Accessed: 2021-04-15. 1, 2.1

[139] Gunel Jahangirova and Paolo Tonella. An empirical evaluation of mutation operators for deep learning systems. In *International Conference on Software Testing, Validation and Verification*, ICST '20, pages 74–84. IEEE, 2020. 2.1

[140] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Quality metrics and oracles for autonomous vehicles testing. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 5.3.1

[141] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010. 1, 2.1, 5, 5.3, 5.3.3

[142] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, ICSE '13, pages 672–681, 2013. 4.4.3

[143] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967. 4.2, 4.3.2

[144] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018. 2.2

[145] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440, 2014. 3.2

[146] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *International Symposium on Foundations of Software Engineering*, FSE'14, pages 654–665, 2014. 1

[147] Tamer Kahveci and Ambuj Singh. Variable length queries for time series data. In *International Conference on Data Engineering*, pages 273–282. IEEE, 2001. 4.2

[148] Tamer Kahveci, Ambuj Singh, and Aliekber Gurel. Similarity searching for multi-attribute sequences. In *International Conference on Scientific and Statistical Database Manage-*

ment, pages 175–184. IEEE, 2002. 4.2

[149] Aaron Kane, Thomas Fuhrman, and Philip Koopman. Monitor based oracles for cyber-physical system testing: Practical experience report. In *International Conference on Dependable Systems and Networks*, DSN '14, pages 148–155, 2014. 1, 2.1, 4

[150] Mubbasir Kapadia, Matt Wang, Shawn Singh, Glenn Reinman, and Petros Faloutsos. Scenario space: characterizing coverage, quality, and failure of steering algorithms. In *Eurographics symposium on computer animation*, pages 53–62, 2011. 1, 2.1, 5, 5.4.2

[151] Christian Kästner, Alexander Von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *International Workshop on Feature-Oriented Software Development*, pages 1–8. ACM, 2012. 3.2.3, 5.2

[152] Leonard Kaufmann and Peter Rousseeuw. Clustering by means of medoids. *Data Analysis based on the L1-Norm and Related Methods*, pages 405–416, 01 1987. 4.2, 4.2

[153] Eamonn Keogh. Exact indexing of dynamic time warping. In *International Conference on Very Large Data Bases*, VLDB '02, pages 406–417, 2002. 4.2

[154] Alif Ridzuan Khairuddin, Mohamad Shukor Talib, and Habibollah Haron. Review on simultaneous localization and mapping (SLAM). In *International Conference on Control System, Computing and Engineering*, ICCSCE '15, pages 85–90. IEEE, 2015. 5.4.2

[155] Siddhartha Kumar Khaitan and James D McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, 2014. 2.1

[156] Manju Khari. Empirical evaluation of automated test suite generation and optimization. *Arabian Journal for Science and Engineering*, pages 1–17, 2019. 1, 2.1, 5, 5.3

[157] Barbara A. Kitchenham and Shari L. Pfleeger. Principles of survey research: Parts 1 – 6. *Software Engineering Notes*, 1995 and 1996. 3.3.1, 3.3.1

[158] Florian Klück, Yihao Li, Mihai Nica, Jianbo Tao, and Franz Wotawa. Using ontologies for test suites generation for automated and autonomous driving functions. In *International Symposium on Software Reliability Engineering Workshops*, ISSREW '18, pages 118–123. IEEE, 2018. 3.3.2, 5.2

[159] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *International Conference on Intelligent Robots and Systems*, volume 3 of *IROS '04*, pages 2149–2154. IEEE, 2004. 2.2, 5.1, 5.2, 1

[160] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. It takes a village to build a robot: An empirical study of the ROS ecosystem. In *International Conference on Software Maintenance and Evolution*, ICSME '20, 2020. 2.2, 5.1

[161] Terry K. Koo and Mae Y. Li. A Guideline of Selecting and Reporting Intraclass Correlation Coefficients for Reliability Research. *Journal of Chiropractic Medicine*, 15(2): 155–163, 2016. 4.4.4

[162] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017. 1, 2.1, 3.1.3

[163] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007. 4.2

[164] Rakesh Kumar and Jyotishree. Blending roulette wheel selection & rank selection in genetic algorithms. *International Journal of Machine Learning and Computing*, 2(4): 365–370, 2012. 5.4.3

[165] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *International Symposium on Software Testing and Analysis*, ISSTA '19, pages 101–111, 2019. 2.1, 5.3, 5.3.2

[166] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Validation and Verification*, ICST '19, pages 312–322. IEEE, 2019. 2.1, 5.3, 5.3.2

[167] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *International Conference on Software Engineering*, ICSE '20, pages 1471–1482, 2020. 2.1, 5.3, 5.3.2

[168] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*, 41(12): 1236–1256, December 2015. ISSN 0098-5589. 3.2, 3.2.1

[169] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2016. 3.1.3

[170] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016. 1, 2.1, 3.3.3, 5, 5.3.2

[171] Joel Lehman and Kenneth O Stanley. Novelty search and the problem with objectives. In *Genetic programming theory and practice IX*, pages 37–56. Springer, 2011. 5.3.3

[172] Caroline Lemieux. Mining temporal properties of data invariants. In *International Conference on Software Engineering*, ICSE '15, pages 751–753, 2015. 1, 2.1

[173] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL specification mining. In *International Conference on Automated Software Engineering*, ASE '15, pages 81–92, 2015. 1, 2.1, 4.5

[174] John J Leonard and Hugh F Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on robotics and Automation*, 7(3):376–382, 1991. 5.1

[175] LG. LGSVL Simulator. `https://www.lgsvlsimulator.com`. Accessed: 2021-04-15. 2.2

[176] Husheng Li. *Communications for control in cyber physical systems: theory, design and applications in smart grids*, chapter 1-Introduction to cyber physical systems. Morgan

Kaufmann, 2016. 1, 2.1, 3.1.2

[177] J Jenny Li, David Weiss, and Howell Yee. Code-coverage guided prioritized test generation. *Information and Software Technology*, 48(12):1187–1198, 2006. 2.1, 5.3, 5.4.2

[178] W. Li, C. W. Pan, R. Zhang, J. P. Ren, Y. X. Ma, J. Fang, F. L. Yan, Q. C. Geng, X. Y. Huang, H. J. Gong, W. W. Xu, G. P. Wang, D. Manocha, and R. G. Yang. AADS: Augmented autonomous driving simulation using data-driven algorithms. *Science Robotics*, 4 (28), 2019. 2.2

[179] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. Metamorphic model-based testing of autonomous systems. In *International Workshop on Metamorphic Testing*, MET '17, pages 35–41, 2017. 2.1, 2.2, 3.1.3, 4.5

[180] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *Computer Architecture News*, 45(1):677–691, 2017. 3.3.3

[181] S. Lloyd. Least squares quantization in pcm. *Transactions on Information Theory*, 28(2): 129–137, 2006. 4.2

[182] Pablo Loyola, Matt Staats, In-Young Ko, and Gregg Rothermel. Dodona: automated oracle data set selection. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 193–203, 2014. 1, 2.1, 4

[183] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys*, 52(5):100, 2019. 1, 2.1, 3.1.3, 3.3

[184] Ingo Lütkebohle. Determinism in robotics software. `https://roscon.ros.org/2017/presentations/ROSCon%202017%20Determinism%20in%20ROS.pdf`, 2017. Accessed: 2021-04-15. 1, 2.1, 5, 5.3, 5.3.2

[185] M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. *IEEE Transactions on Reliability*, 43(4):527–535, 1994. 2.1, 5.3

[186] Kathleen M. MacQueen, Eleanor McLellan-Lemal, Kelly Bartholow, and Bobby Milstein. *Team-based Codebook Development: Structure, Process, and Agreement*, pages 119–135. Rowman Altamira, 2008. 3.3.1, 3.3.1

[187] Rupak Majumdar, Aman Mathur, Marcus Pirron, Laura Stegner, and Damien Zufferey. Paracosm: A language and tool for testing autonomous driving systems. *arXiv preprint arXiv:1902.01084*, 2019. 3.3.2, 5.2

[188] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *International Conference on Software Engineering*, ICSE '13, pages 1012–1021, 2013. 1, 2.1, 4

[189] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947. 4.1, 5.3.1

[190] Brian Marick. How to misuse code coverage. In *Interational Conference on Testing Computer Software*, pages 16–18, 1999. 5.3

[191] Brian Marick. Faults of omission. *Software Testing and Quality Engineering Magazine*, 2(1), 2000. 5.3

[192] Dusica Marijan, Arnaud Gotlieb, and Mohit Kumar Ahuja. Challenges of testing machine learning based systems. In *International Conference On Artificial Intelligence Testing*, AITest'19, pages 101–102. IEEE, 2019. 1, 2.1, 2.1

[193] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1): 176–205, Feb 2015. ISSN 1573-7616. 3.2.1, 3.2.3

[194] Wes Masri and Rawad Abou Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM transactions on software engineering and methodology*, 23(1):1–28, 2014. 5.3

[195] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *International Workshop on Defects in Large Software Systems*, pages 1–5, 2009. 5.3

[196] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951. 5.3.1

[197] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *International Conference on Software Engineering*, ICSE'16, pages 595–606, 2016. 1, 2.1, 4.1.1, 5, 5.4

[198] Phil McMinn. Search-based software testing: Past, present and future. In *International Conference on Software Testing, Verification and Validation Workshops*, ICST'11, pages 153–163. IEEE, 2011. 2.1, 5.4

[199] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *International Workshop on Software Test Output Validation*, pages 1–4, 2010. 1, 2.1

[200] Carter McNamara. General guidelines for conducting interviews, 1999. 3.1.1

[201] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'19, pages 27–38, 2019. 2.1

[202] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *Intelligent Vehicles Symposium*, pages 1821–1827. IEEE, 2018. 1, 2.1, 5, 5.4.2

[203] John Micco. Flaky tests at Google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, May 2016. Accessed: 2021-04-15. 3.3.2

[204] Seyedali Mirjalili. Genetic algorithm. In *Evolutionary algorithms and neural networks*, pages 43–55. Springer, 2019. 5.4

[205] Swarup Mohalik, Ambar A Gadkari, Anand Yeolekar, KC Shashidhar, and S Ramesh. Automatic test case generation from simulink/stateflow models using model checking.

International Conference on Software Testing, Verification and Reliability, 24(2):155–180, 2014. 1, 2.1, 4.1.1, 5, 5.4

[206] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021. 5.4.2

[207] Morten Mossige, Arnaud Gotlieb, and Hein Meling. Testing robot controllers using constraint programming and continuous integration. *Information and Software Technology*, 57:169–185, 2015. 3.1.3

[208] Galen E Mullins, Paul G Stankiewicz, R Chad Hawthorne, and Satyandra K Gupta. Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles. *Journal of Systems and Software*, 137:197–215, 2018. 2.1, 5.4.2

[209] Tadahiko Murata and Hisao Ishibuchi. Moga: Multi-objective genetic algorithms. In *International Conference on Evolutionary Computation*, volume 1, pages 289–294, 1995. 3

[210] Patric Nader, Paul Honeine, and Pierre Beauseroy. l_p-norms in one-class classification for intrusion detection in scada systems. *Transactions on Industrial Informatics*, 10(4): 2308–2317, 2014. 2.1, 4.3.1

[211] Ulrich Nehmzow. *Mobile robotics: a practical introduction*. Springer Science & Business Media, 2012. 5.1

[212] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. A preliminary survey on software testing practices in Australia. In *Australian Software Engineering Conference*, pages 116–125. IEEE, 2004. 1, 2.1

[213] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering*, ICSE '14, pages 608–619, 2014. 1, 2.1

[214] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. Syminfer: inferring program invariants using symbolic states. In *International Conference on Automated Software Engineering*, ASE'17, pages 804–814, 2017. 1, 2.1

[215] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, ICSE '93, pages 100–107, 1993. 2.1, 5.3.1

[216] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *International Conference on Automated Software Engineering*, ASE '14, pages 19–30, 2014. 1, 2.1, 4.5

[217] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, and Marko Bertogna. F1/10: An open-source autonomous cyber-physical platform, 2019. 2.2, 4.4.7

[218] Open Robtics. About ROS. `https://www.ros.org/about-ros/`. Accessed: 2021-04-15. 2.2

[219] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Phriky-units: a lightweight, annotation-free physical unit inconsistency detection tool. In *International Symposium on Software Testing and Analysis*, ISSTA '17, pages 352–355, 2017. 4.6

[220] John-Paul W. Ore. *Dimensional Analysis of Robot Software without Developer Annotations*. PhD thesis, University of Nebraska-Lincoln, 7 2019. 4.6

[221] Ali Paikan, Silvio Traversaro, Francesco Nori, and Lorenzo Natale. A generic testing framework for test driven development of robotic systems. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 216–225. Springer, 2015. 3.1.3

[222] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *International Conference on Software Engineering*, ICSE'18, pages 537–548. IEEE, 2018. 1, 2.1, 5, 5.3, 5.3.3

[223] Sangmin Park, BM Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: Achieving higher statement coverage faster. In *International Symposium on the Foundations of Software Engineering*, FSE '12, pages 1–11, 2012. 2.1, 5.3, 5.4.2

[224] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design. In *Conference on Decision and Control and European Control Conference*, pages 2195–2201, 2011. 2.1

[225] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *International Conference on Software Testing, Verification and Validation*, ICST '13, pages 342–351, 2013. 1, 2.1, 4

[226] Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. 4.2

[227] Sasha Pietrzik and Balasubramaniyan Chandrasekaran. Testing autonomous path planning algorithms and setup for robotic vehicle navigation. In *Annual Ubiquitous Computing, Electronics & Mobile Communication Conference*, UEMCON '18, pages 485–488. IEEE, 2018. 2.2, 5.1

[228] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators. In *Annual Conference Towards Autonomous Robotic Systems*, pages 357–368, 2018. 2.2, 3.3

[229] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 2.2, 3.3.1, 4.4.7, 4.5

[230] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985. 2.1, 5.3

[231] Chotirat Ann Ratanamahatana and Eamonn Keogh. Three myths about dynamic time warping data mining. In *International Conference on Data Mining*, ICDM '05, pages

506–510, 2005. 4.3

[232] Danda B Rawat, Joel JPC Rodrigues, and Ivan Stojmenovic. *Cyber-physical systems: from theory to practice*. CRC Press, 2015. 3.1.2, 5, 5.4

[233] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. Survey on scenario-based safety assessment of automated vehicles. *IEEE Access*, 8: 87456–87477, 2020. 1, 2.1, 5, 5.4.2

[234] Clément Robert, Thierry Sotiropoulos, Jérémie Guiochet, Hélène Waeselynck, and Simon Vernhes. The virtual lands of Oz: testing an agribot in simulation. *Empirical Software Engineering*, 2020. 3.3.2

[235] Sheila B. Robinson and Kimberley F. Leonard. *Designing Quality Survey Questions*. SAGE Publications, 1 edition, 2018. 3.3.1, 3.3.1

[236] Rockstar Games. Grand Theft Auto V. `https://www.rockstargames.com/games/info/V`, 2013. Accessed: 2021-04-15. 5.2

[237] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems*, IROS '13, pages 1321–1326, 2013. 2.2

[238] Jarek Rossignac. Shape complexity. *The visual computer*, 21(12):985–996, 2005. 5.6

[239] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987. 4.3.2

[240] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006. 1, 2.1

[241] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, pages 58–66, 2018. 4.4.3

[242] S. K. Sahoo, J. Criswell, and V. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *International Conference on Software Engineering*, ICSE '10, pages 485–494, 2010. 3.2

[243] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019. 2.1

[244] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015. 3.3.1, 3.3.1

[245] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007. 4.2, 4.3.2

[246] Lukas Schmidt, Apurva Narayan, and Sebastian Fischmeister. TREM: a tool for mining timed regular specifications from system traces. In *International Conference on Automated Software Engineering*, ASE '17, pages 901–906, 2017. 2.1

[247] I. Seidman. *Interviewing as Qualitative Research: A Guide for Researchers in Education and the Social Sciences*. Teachers College Press, 2006. ISBN 9780807746660. 3.1.4

[248] Sanjit A Seshia, Shiyan Hu, Wenchao Li, and Qi Zhu. Design automation of cyber-

physical systems: Challenges, advances, and opportunities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1421–1434, 2016. 2.1, 2.1, 3.1.2, 3.1.2

[249] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In Marco Hutter and Roland Siegwart, editors, *Field and Service Robotics*, pages 621–635, 2018. 2.2

[250] Sandy Sheng and Nicole Becker. Challenges in standardizing ram testing for small unmanned robotic systems. In *Reliability and Maintainability Symposium*, RAMS'13, pages 1–6. IEEE, 2013. 3.1.3

[251] Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors. *Guide to Advanced Empirical Software Engineering*. Springer, 2008. 3.1.1, 3.1.4

[252] Susan Elliot Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *International Conference on Software Engineering*, ICSE '03, pages 74–83, 2003. 3.2

[253] Helena Skutkova, Martin Vitek, Petr Babula, Rene Kizek, and Ivo Provaznik. Classification of genomic signals using dynamic time warping. *BMC bioinformatics*, 14(10):S1, 2013. 4.2

[254] Thierry Sotiropoulos, Hélène Waeselynck, and Jérémie Guiochet. Can robot navigation bugs be found in simulation? an exploratory study. In *Software Quality, Reliability and Security*, QRS '17, pages 150–159, 2017. 2.1, 3.1.2, 3.2, 3.3.2

[255] Aaron Staranowicz and Gian Luca Mariottini. A survey and comparison of commercial and open-source robotic simulator software. In *Pervasive Technologies Related to Assitive Environments*, PETRA '11, pages 56:1–56:8, 2011. 2.2, 3.3

[256] Nick Statt. Boston Dynamics' Spot robot is helping hospitals remotely treat coronavirus patients. *The Verge*, 2020. URL `https://www.theverge.com/2020/4/23/21231855/boston-dynamics-spot-robot-covid-19-coronavirus-telemedicine`. Accessed: 2021-04-15. 1

[257] Gerald Steinbauer. A survey about faults of robots used in robocup. In Xiaoping Chen, Peter Stone, Luis Enrique Sucar, and Tijn van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 344–355. Springer Berlin Heidelberg, 2013. 3.2

[258] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In *International Conference on Software Engineering*, ICSE '20, pages 359–371, 2020. 2.1

[259] S. K. S. Sze and M. R. Lyu. Atacobol-a cobol test coverage analysis tool and its applications. In *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, pages 327–335, 2000. 2.1, 5.3

[260] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *International Conference on Software Engineering*, ICSE '17 Poster, pages 180–182, 2017. 3.2

[261] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011. 1

[262] Andreas Theissler. Detecting known and unknown faults in automotive systems using ensemble-based anomaly detection. *Knowledge-Based Systems*, 123:163–173, 2017. 2.1

[263] Zoe Thomas. Coronavirus: Will COVID-19 speed up the use of robots to replace human workers? *BBC News*, 2019. URL `https://www.bbc.com/news/technology-52340651`. Accessed: 2021-04-15. 1

[264] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering*, ICSE '18, pages 303–314, 2018. 2.1, 3.1.3

[265] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *International Conference on Software Testing, Verification and Validation*, ICST '18, pages 331–342. IEEE, 2018. 1, 1.2, 3, 3.1.2, 3.2, 3.3.2

[266] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. 2.2

[267] Tony Tolker-Nielsen and ESA Inspector General. Schiaparelli anomaly inquiry. Technical report, The European Space Agency, 2017. 1

[268] Cumhur Erkan Tuncali. *Search-based Test Generation for Automated Driving Systems: From Perception to Control Logic*. PhD thesis, Arizona State University, 2019. 1, 2.1, 4.1.1, 5

[269] Cumhur Erkan Tuncali, Theodore P Pavlic, and Georgios Fainekos. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *International Conference on Intelligent Transportation Systems*, ITSC'16, pages 1470–1475. IEEE, 2016. 1, 2.1, 4.1.1, 5

[270] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *International Conference on Intelligent Transportation Systems*, pages 982–988. IEEE, 2015. 2.1, 2

[271] Rijnard van Tonder and Claire Le Goues. Lightweight multi-language syntax transformation with parser parser combinators. In *Conference on Programming Language Design and Implementation*, PLDI '19, pages 363–378, 2019. 5.3.1

[272] Vandi Verma, Geoff Gordon, Reid Simmons, and Sebastian Thrun. Real-time fault diagnosis [robot fault diagnosis]. *Robotics & Automation Magazine*, 11(2):56–66, 2004. 2.1

[273] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *International Conference on Software Engineering*, ICSE '09, pages 45–55. IEEE, 2009. 5.3

[274] Westley Weimer, Stephanie Forrest, Claire Le Goues, and Miryung Kim. Cooperative, trusted software repair for cyber physical system resiliency. Technical report, University of Virginia Charlottesville United States, 2018. 2.2, 4.5

[275] Johannes Wienke and Sebastian Wrede. Results of the survey: failures in robotics and intelligent systems. *arXiv preprint arXiv:1708.07379*, 2017. 2.1

[276] Johannes Wienke, Sebastian Meyer zu Borgsen, and Sebastian Wrede. A data set for fault detection research on component-based robotic systems. In *Towards Autonomous Robotic Systems*, TAROS '16, pages 339–350, 2016. ISBN 978-3-319-40379-3. 3.2

[277] Andreas Windisch. Search-based testing of complex simulink models containing stateflow diagrams. In *International Conference on Software Engineering*, ICSE'09, pages 395–398. IEEE, 2009. 1, 2.1, 4.1.1, 5, 5.4

[278] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on autonomous mobile service robots*, 2016. 5.1

[279] Keenan Wyrobek. The origin story of ROS, the Linux of robotics. `https://spectrum.ieee.org/automaton/robotics/robotics-software/the-origin-story-of-ros-the-linux-of-robotics`, 2017. 3.3.1, 4.4.7

[280] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *Software Analysis, Evolution and Reengineering*, SANER '17, pages 138–147, 2017. 3.3.2

[281] Qin Xia, Jianli Duan, Feng Gao, Qiuxia Hu, and Yingdong He. Test scenario design for intelligent driving system ensuring coverage and effectiveness. *International Journal of Automotive Technology*, 19(4):751–758, 2018. 1, 2.1, 5, 5.4.2, 5.4.2, 5.4.2, 5.4.2

[282] Qinghua Xu, Shaukat Ali, and Tao Yue. Digital twin-based anomaly detection in cyber-physical systems. In *International Conference on Software Testing, Validation and Verification*, ICST '21. IEEE, April 2021. 2.1

[283] Rui Xu and Don Wunsch. *Clustering*, volume 10. John Wiley & Sons, 2008. 4.2

[284] Shenao Yan, Guanhong Tao, Xuwei Liu, Juan Zhai, Shiqing Ma, Lei Xu, and Xiangyu Zhang. Correlations between deep neural network model coverage criteria and model quality. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, pages 775–787, 2020. 2.1

[285] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, 2006. 1, 2.1

[286] Kiyoung Yang and Cyrus Shahabi. A pca-based similarity measure for multivariate time series. In *International Workshop on Multimedia Databases*, MMDB '04, pages 65–74, 2004. 4.2, 4.2

[287] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009. 2.1, 5.3, 5.4.2

[288] Nong Ye, Syed Masum Emran, Qiang Chen, and Sean Vilbert. Multivariate statistical analysis of audit trails for host-based intrusion detection. *Transactions on computers*, 51 (7):810–820, 2002. 2.1, 3.1.3, 4.5

[289] Man Zhang, Shaukat Ali, Tao Yue, and Roland Norgren. Uncertainty-wise evolution of test ready models. *Information & Software Technology*, 87:140–159, 2017. 3.1.3

[290] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *International Conference on Automated Software Engineering*, ASE '18, pages 132–142, 2018. 3.1.3

[291] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. On the state of the art in verification and validation in cyber physical systems. *The University of Texas at Austin, The Center for Advanced Research in Software Engineering, Tech. Rep. TR-ARiSE-2014-001*, 1485, 2014. 1, 2.1, 3.1.2, 3.1.2

[292] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. Perceptions on the state of the art in verification and validation in cyber-physical systems. *IEEE Systems Journal*, 11(4):2614–2627, Dec 2017. 1, 3.3

[293] Vivienne Jia Zhong, Rolf Dornberger, and Thomas Hanne. Comparison of the behavior of swarm robots with their computer simulations applying target-searching algorithms. *International Journal of Mechanical Engineering and Robotics Research*, 2018. 1

[294] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, February 2019. 2.1, 3.1.3

[295] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009. 4.2

[296] Ehsan Zibaei, Sebastian Banescu, and Alexander Pretschner. Diagnosis of safety incidents for cyber-physical systems: A uav example. In *International Conference on System Reliability and Safety*, ICSRS'18, pages 120–129. IEEE, 2018. 1, 2.1, 2.1, 2.2, 3.1.3, 4, 4.5