Advancing Cyber-Physical Systems Testing with Machine Learning: Effective, Reliable and Interpretable Approaches

Baharin Aliashrafi Jodat

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Cyber-Physical Systems (CPS) are complex integrations of computational and physical processes, deployed in domains such as autonomous driving, aerospace, and networking. Traditional exhaustive testing for CPS is typically infeasible because the space of system states and environmental conditions grows combinatorially. Simulation-based testing offers a more scalable and controlled alternative, but it raises concerns about both effectiveness and reliability. Key issues include the computational cost and inherent flakiness of CPS simulators, along with the difficulty of generating valid and realistic test inputs. Addressing these issues is crucial to ensure that testing not only detects failures but also yields actionable insights for system validation and debugging. This requires interpretable testing outcomes so that engineers can understand the rationale behind test verdicts and use them to guide root-cause analysis, requirements refinement, and design improvements.

This thesis proposes a set of novel, data-driven approaches that integrate search-based software engineering with machine learning to enhance the effectiveness and reliability of CPS testing. First, we develop approaches based on machine learning for test generation. These approaches explore and exploit the search space for identifying diverse system behaviours and exposing failures that lie in boundary regions. Second, we develop a novel surrogate-assisted test generation technique that reduces dependence on computationally expensive simulators to predict test outcomes while preserving predictive accuracy. Third, we introduce interpretable approaches that provide engineers with clear explanations of the conditions that lead to different system behaviours, such as passing, failing, and non-robust behaviours. Our approaches for explanation are based on interpretable machine learning

models and genetic programming. Explanations provided by our approaches are not only accurate but are also interpretable, allowing engineers to easily understand them. Further, our explanations are minimally impacted by the flakiness in the datasets used to infer the explanations, meaning that they remain stable and reliable and thus offer consistent, actionable insight into system behaviour despite underlying non-determinism.

Across case studies in networking, autonomous driving, and industrial-scale Simulink models, our empirical results show that these techniques are accurate and can uncover different system behaviours, such as passing, failing, and non-robust behaviours, while reducing the need for expensive simulator executions. Our results further indicate that the learned artifacts, including failure characterizations, explanations, and automated validators, remain reliable enough to support practical engineering tasks such as triaging flaky outcomes, filtering uninformative tests, and guiding debugging and validation decisions.

Together these contributions enhance CPS testing by improving efficiency, reliability, and interpretability. By systematically generating test cases, drastically reducing computational costs and providing human-understandable explanations for system behaviours, this thesis transforms simulation-based testing from a largely black-box activity into a rigorous, efficient, and insightful engineering process.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Shiva Nejati, for her mentorship and support throughout this work. Our discussions and her thoughtful feedback shaped the direction of this thesis and helped me grow as a researcher. Among the many lessons she taught me, two stand out and will continue to guide my work: *writing is debugging the idea* and *be your first critical reader.*

I extend my sincere gratitude to Dr. Mehrdad (Mike) Sabetzadeh for his constructive feedback, attention to detail, and critical insights, which significantly strengthened this thesis. I am grateful to my examining committee, Dr. Sebastiano Panichella, Dr. Nafiseh Kahani, and Dr. Amy Felty, for their time, careful reading of my thesis, and thoughtful questions.

I would like to thank my colleagues at the Sedna Lab, with whom I shared countless discussions and conference experiences that enriched both my research and personal growth.

Lastly, I would like to express my deepest gratitude to my family: to my mother, Roghieh, for being a great researcher herself and showing me the value of curiosity and dedication; to my father, Rajab, for demonstrating how perseverance and hard work can achieve great things; to my sister, Yasamin, for being my role model since childhood and for showing me the value of continuous improvement; and to my husband, Mohammadreza Mohajer, for being my best friend and for his unwavering love, patience, and understanding throughout this process. His support and constant encouragement gave me the strength to move forward, even during the most challenging times.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Cyber-physical systems (CPS) are advanced integrations of computing and physical processes, allowing for seamless interaction between the digital and physical worlds. CPS are widely used in areas such as autonomous vehicles, unmanned aerial vehicles, healthcare, manufacturing, and telecommunications [BG11, SWYS11, KPT23]. Testing CPS is crucial because their complexity and growing use in safety-critical tasks mean that failures can be catastrophic, which requires rigorous assessment of reliability and performance [SKDB22, ads25].

Traditional exhaustive testing for CPS is often extremely time-consuming and costly [KDJ+16]. Exhaustive methods require evaluating numerous scenarios and configurations, which leads to substantial resource expenditure and delays in the development

cycle [KDJ+16]. For example, testing an autonomous driving system under different real-world conditions – such as changes in weather, road surfaces, or the behaviour of pedestrians and other drivers – would require millions of kilometres of physical driving, which is impossible within a reasonable timeframe. Similarly, testing an autopilot system across many possible flight configurations and weather conditions would require building and crashing thousands of aircraft for every possible scenario, which is prohibitively expensive and practically impossible.

Simulation-based testing has therefore emerged as a powerful alternative to traditional testing. In simulation-based testing, engineers create a virtual environment that replicates key components of a system, including its hardware and software. This environment provides a testing ground for evaluating system performance before real-world deployment [BAN+21,raq25,BKR+25]. For autonomous driving, simulators such as BeamNG [bea25] and Carla [DRC+17] enable generation of diverse combinations of traffic and weather patterns and road layouts in a controlled and repeatable manner. For aircraft autopilot systems, flight simulators such as X-Plane [xpl25] and MATLAB/Simulink-based environments [aut25] model aircraft dynamics and sensor data, enabling a wide range of flight scenarios to be exercised.

Building on such applications, simulation-based testing of CPS has recently been used for finding test inputs that reveal failures in SUT (System Under Test) [KPT24, BT24, BKD+23, BGK+23]. Beyond failure detection, simulation-based techniques have been used to uncover unknown environmental assumptions that constrain the operating conditions of the SUT [GMN+21], to stress test cloud-based CPS aiming to identify performance bottlenecks and ensure robustness under high load or adverse conditions [LMN+24] and to

support security analysis, including the generation and evaluation of attack scenarios in networked CPS [ANSS24].

A high-level overview of a typical simulation-based testing workflow for CPS is presented in Figure 1.1. The process begins with a test-input generator that produces a set of test inputs (Step 1 in Figure 1.1). These test inputs can be generated randomly or through search-based and learning-based techniques, depending on the testing objectives. The generated tests are then executed on a simulator (Step 2 in Figure 1.1). The simulator is implemented to represent the real CPS for testing purposes and models the aspects needed for test execution, including the system's dynamics, relevant environmental conditions, and the behaviour of sensors and actuators. As the CPS evolves, the simulator can be updated to remain consistent with it and preserve fidelity. After execution, a test oracle evaluates each test result and assigns a pass or fail verdict to each test input (Step 3 in Figure 1.1). The oracle's decisions are grounded in CPS requirements defined by domain expertise, official documentation, or applicable standards and regulations. Test oracles can range from human experts to assertions, formal specifications, or ML-based mechanisms. Finally, the obtained verdicts and execution data are leveraged for different objectives such as fault detection, root cause analysis, robustness assessment, requirements conformance checking, and regression testing (Step 4 in Figure 1.1). To assess such workflows and compare different testing approaches, we focus on two key attributes of the testing process: its *effectiveness* and its *reliability*.

In software testing terminology, *effectiveness* is a multi-faceted attribute that combines capability, interpretability, and efficiency. Capability captures the extent to which a testing method generates meaningful and diverse test inputs that exercise the system, reveal

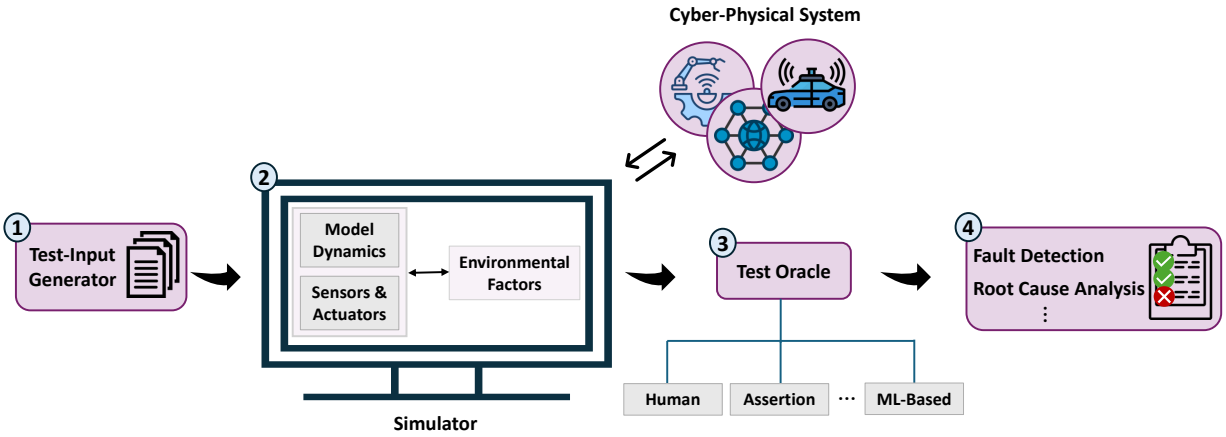


Figure 1.1: A typical workflow of simulation-based testing of CPS

failures, and validate system behaviour across a broad range of operational conditions. Interpretability ensures that engineers not only identify system failures but also can diagnose root causes of such failures. This ability is essential for guiding corrective measures and informing system improvements, ultimately reducing the likelihood of similar issues recurring. Efficiency ensures that capability and interpretability are achieved within reasonable cost and time budgets, which is particularly critical in resource-intensive testing environments.

For example, in the context of an aircraft autopilot system, effectiveness goes beyond merely detecting that the system fails to maintain stable flight. An effective testing method should provide interpretable explanations for the reasons behind failures. Such explanations may reveal whether the issue stems from a malfunctioning sensor, a flaw in the internal decision-making algorithms, or an external factor such as severe turbulence that exceeded the system's tolerance thresholds. Providing this level of insight helps determine whether engineers should replace hardware, refine algorithms, or improve robustness to

environmental disturbances. Without interpretability, test outcomes only indicate that a failure occurred but offer limited guidance for remediation, which undermines the effectiveness of the testing. Efficiency then determines whether these insights can be obtained at an acceptable cost and within realistic time limits.

Complementing effectiveness, *reliability* focuses on trustworthiness of test verdicts (pass or fail). A reliable testing process should be robust to factors such as simulator flakiness, nondeterminism, and environmental variability, and it should exercise CPS under conditions that accurately reflect real-world operating scenarios. For instance, testing an autonomous driving system on roads with overly sharp curves, unrealistic designs and traffic patterns provides limited insight into how the system will actually perform in the real world. Reliable test outcomes reduce the likelihood of false positives, where failures are incorrectly reported, and false negatives, where actual failures go undetected. Ultimately, reliability increases confidence that insights from simulations and experiments are meaningful, reproducible, and indicative of real CPS behaviour.

In this thesis, we aim to enhance both the effectiveness and reliability of CPS testing by proposing approaches that integrate search-based testing with ML. We leverage ML models to accelerate test generation by accurately predicting the outcomes for test inputs while ensuring that the generated inputs remain realistic and representative of real-world scenarios. Further, we infer explanations for test verdicts that are interpretable and minimally affected by flakiness or nondeterminism. To demonstrate the applicability and impact of our contributions, we conduct case studies in the domains of networking, autonomous driving, and industrial-scale Simulink models. These systems span diverse and representative

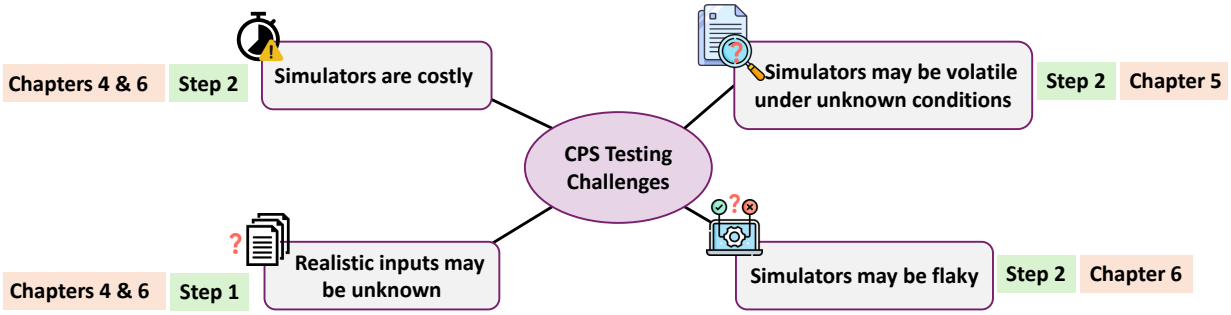


Figure 1.2: CPS testing challenges, the workflow steps in Figure 1.1 where the challenges arise, and the corresponding thesis chapters addressing each challenge

CPS applications and illustrate how our methods improve the effectiveness and reliability of CPS testing.

1.2 Challenges

The simulation-based testing workflow for CPS in Figure 1.1, faces several challenges that can compromise the effectiveness and reliability of the testing process. These challenges stem from generating realistic and meaningful test inputs and from executing those inputs on compute-intensive and potentially flaky simulators. Figure 1.2 provides an overview of these challenges and indicates the specific steps in Figure 1.1 where each one emerges. Below, we describe each challenge in detail and identify gaps in current testing methodologies that motivate the methods developed in this thesis.

Challenge 1: *CPS simulators are typically compute-intensive and executing a test input on these simulators requires substantial time and resources.* This challenge relates to Step 2 in Figure 1.1 where test inputs are executed on the simulator. CPS simulators

are computationally expensive because they must capture complex interactions between physical processes and embedded software, often by solving differential equations for physical phenomena such as motion or fluid dynamics while emulating real-time control and communication. High-fidelity simulations require fine-grained time steps and rich state representations, which further significantly increase the computational load. As a result, executing a single test input can be time-consuming, and evaluating a large test suite may demand substantial computational infrastructure [BKR+25].

To mitigate these costs, prior work has proposed surrogate models that predict whether a test input passes or fails without executing the simulator [Jin05, BANBS16, NSS+23, MNB17]. Such models act as fast approximations of the simulator output and can reduce the number of expensive runs. However, most existing work focuses on predicting verdicts alone. For CPS, merely detecting a failure without providing insights into its root cause yields limited value for diagnosis and design improvement.

Challenge 2: *CPS simulators may be flaky, leading to inconsistent test outcomes.* This challenge is related to Step 2 in Figure 1.1 where test inputs are executed on the simulator. CPS simulators may be non-deterministic due to inherent environmental variability, unpredictable hardware-software interactions, and underlying stochastic processes. Consequently, the outputs for a given test input may vary across different executions which affects the reliability of testing. This issue, which has been observed in simulators for autonomous vehicles, drones, network systems, and other similar systems, leads to flakiness in the test outcomes [BKB+23, NHG21, KPT24, ANN24]. A flaky test is one that non-deterministically passes or fails across different executions.

Surrogate models or verdict explanations that are learned from datasets containing flaky tests may produce less stable and reliable verdict predictions, showing reduced robustness – that is, a reduced ability to provide consistent verdicts or explanations for the same inputs. Current studies suggest running a test input multiple times to determine whether it passes or fails and then discarding flaky tests to avoid inaccuracies [KPT24, BKB+23, ANN24]. However, repeated execution is costly in the CPS setting, given the high cost of a single simulator execution (Challenge 1).

Challenge 3: *Valid and realistic inputs for CPS are unknown.* This challenge relates to Step 1 in Figure 1.1, where test inputs are automatically generated. For CPS with vast, multidimensional input spaces, many generated inputs do not meaningfully exercise the SUT. Some violate SUT's requirements preconditions causing tests to pass or fail regardless of the system's behaviour; others lie outside the system's operational design domain (ODD) [Pra21] or correspond to inherently low-risk, nominal scenarios in which the system requirement is trivially satisfied. Such inputs waste testing resources and distort confidence in the system's dependability.

Existing standards and specifications often describe preconditions and ODDs qualitatively and, when they provide numeric information, it typically appears as broad ranges for system inputs. These ranges rarely have the precision needed to judge whether a candidate test input is valid and realistic for a specific requirement, since the requirement-specific ranges are often narrower and depend on input combinations. For example, for an aircraft autopilot system, the input ranges are $[-1, 1]$ radians for the pitchwheel signal (which controls elevator deflection) and $[0, 100\%]$ for the throttle (which controls engine thrust). For a proper ascent, however, the valid ranges are $[0, 1]$ radians for the pitchwheel and

$[50\%, 100\%]$ for the throttle. Moreover, system specifications rarely capture how combinations of inputs or interdependencies between them define the valid operational domain, since such interactions are often too complex or numerous to describe manually. Consequently, relying solely on broad input ranges is insufficient for generating valid and realistic test inputs, and automated test input validation becomes crucial for improving the reliability of test results [RT23].

Traditional testing approaches rely on formal assume-guarantee and design-by-contract techniques to ensure that test inputs adhere to the system requirements. However, these techniques require high-level formal system specifications which may not exist for many real-world CPS. Further, recent studies explore test input validity for deep learning (DL) models [HMC+22, RT23]. These studies perform human subject experiments and establish metrics that determine test input validity for DL models. However, our case studies focus on CPS, where the inputs are numeric in nature, requiring careful consideration of the range and distribution of these values.

Challenge 4: *CPS simulators may be volatile under unknown conditions.* This challenge is related to Step 2 in Figure 1.1 where test inputs are executed on the simulator. CPS can be sensitive to perturbations in their input, caused by, among other factors, uncertainty in the environment, evolving system-usage patterns, internal computation errors, and network degradation. Under such conditions, the system becomes volatile or non-robust, causing the output of the system to unexpectedly change from passing to failing or vice versa [ALFS11]. This volatility reduces the effectiveness of testing, since it becomes difficult to determine whether observed failures stem from genuine design flaws or from inherently unstable operating regions. Identifying and characterizing these non-robust re-

gions is therefore important for focusing testing effort on inputs that are informative for robustness assessment.

Recent studies suggest that injecting faults into the system – such as through code mutation [LMX05], electromagnetic interference [HHS+11], and power supply disturbances [LL08] – can help evaluate the system's robustness and identify conditions under which the system is volatile. However, these approaches often face challenges due to the black-box nature of many CPS simulators, which limits access to the underlying code and hardware components.

1.3 Methodology

In this section, we provide a high-level overview of the methods developed in this thesis, which build upon techniques from search-based software engineering (SBSE), ML, and empirical research. We outline a common structure, illustrated in Figure 1.3, that forms the basis for the methods presented in subsequent chapters. This structure consists of three main steps: (1) exploring or exploiting the input search space, (2) approximating test outputs, and (3) inferring explanations for test outputs. Below, we discuss each step.

(1) Explore/Exploit Input Search Space. Given a CPS and its input search space, the first step in Figure 1.3 focuses on exploring and exploiting this space. *Exploration* refers to broadly sampling the input space with the goal of uncovering diverse system behaviours. *Exploitation* involves directing the search toward regions of the input space that are particularly interesting, such as the boundary regions between passing and failing

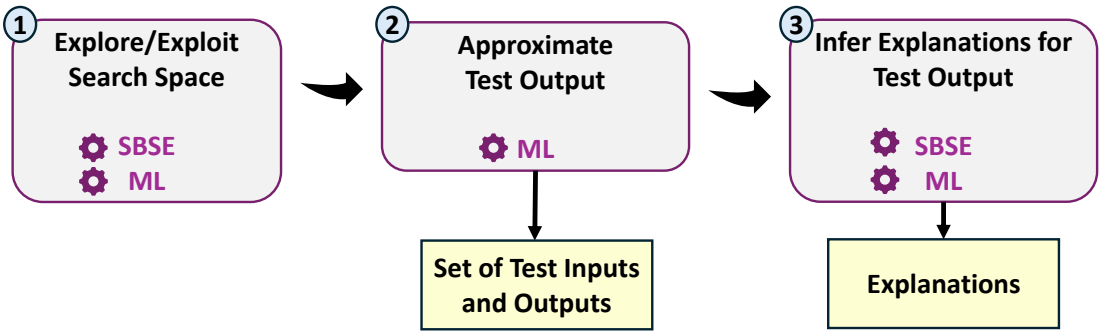


Figure 1.3: A high-level overview of the key components in the methods developed in this thesis for improving CPS testing

tests, as well as regions where the system exhibits non-robust behaviours. In this thesis, we employ SBSE techniques such as adaptive random search for explorative search and use ML models to guide exploitative search toward informative regions where failures or non-robust behaviours are most likely to occur.

(2) Approximate Test Output. Once candidate test inputs are generated in Step 1 of Figure 1.3, the next step is to determine their outputs. As discussed in Challenge 1 in Section 1.2, executing CPS simulators at scale is computationally expensive. To reduce this cost, we use ML models as surrogates that approximate simulator behaviour. These surrogate models, trained on a representative subset of simulator executions, can predict test outcomes with significantly lower cost. We investigate various ML models, including random forests and neural networks, and study their accuracy and efficiency as surrogates.

(3) Infer Explanation for Test Output. As mentioned in Section 1.1, providing interpretable and accurate explanations for test outcomes is critical for CPS. In Step 3 of Figure 1.3, we propose two approaches to infer explanations: (1) genetic programming (GP) and (2) ML models such as decision trees and decision rules. These methods infer

logical constraints that characterize conditions under which a system is likely to pass, fail or exhibit non-robust behaviours.

Based on Figure 1.3, the output of our methods is twofold. First, our methods efficiently generate accurate *sets of test inputs and their corresponding outputs*, which are useful for system validation and regression testing and cover a wide range of CPS behaviours. Second, our methods generate *explanations of system behaviour*, including insights into passing, failing, and non-robust behaviours. These explanations support engineers in diagnosing root causes, improving system design, and making informed decisions about corrective actions.

1.4 Research Contributions

In this thesis, we address the challenges described in Section 1.2 that hinder the effective and reliable testing of CPS. Our work spans several representative domains, including networked communication systems, autonomous driving systems (ADS), and industrial-scale Simulink models. For network systems, we developed a simulator in collaboration with RabbitRun Technologies Inc., enabling controlled experimentation with realistic communication dynamics and system behaviours. For ADS, we rely on the BeamNG simulator, a widely adopted open-source tool that provides high-fidelity environments for the testing and validation of autonomous driving functionalities [bea25]. For industrial-scale CPS, we use a publicly available benchmark comprising eleven Simulink models from Lockheed Martin [loc25], which capture diverse applications in aerospace and defence.

We propose novel approaches that build upon and extend methods from SBSE, including metaheuristic search algorithms, ML techniques, and empirical research methods. To validate the effectiveness and demonstrate the applicability of our frameworks across diverse CPS, we conduct a comprehensive empirical analysis, using statistical hypothesis tests (e.g., the Mann-Whitney U test [MW47]) to quantitatively compare the performance of our approaches with baseline and state-of-the-art techniques. We further perform a systematic human-subject study to assess how well our explanations for system behaviours align with descriptions in technical standards and in empirical or expert-validated studies[1].

1.4.1 Research Questions

In this thesis, we extensively investigate several research questions, including, but not limited to, the following:

RQ1. Can we efficiently generate accurate test suites for testing compute-intensive CPS? In this research question, we aim to reduce the cost of test execution when generating test suites for our case study systems. Our contribution within the scope of RQ1 is:

Con1.1 We propose an approach that leverages ML models to predict labels for test inputs. Specifically, we propose a novel dynamic surrogate-assisted algorithm that uses multiple surrogate models during search, and dynamically selects the prediction from the most accurate model.

[1]Ethics approval for our study was granted by the University of Ottawa's Research Ethics Board (file number H-11-25-12283).

Evaluation We empirically evaluate the efficiency and accuracy of the proposed approach. To measure efficiency, we evaluate the size of the generated test suites. To measure accuracy, we assess the correctness of the labels assigned to the tests.

RQ2. How can we provide interpretable and accurate insights into the causes of failures in CPS? In this research question, our goal is to provide interpretable and accurate insights into the failures of the system. Our contributions within the scope of RQ2 are:

Con2.1 We explore explorative and exploitative search strategies to systematically generate test inputs and execute them on the SUT, thereby uncovering potential failure scenarios.

Con2.2 We develop an approach to infer failure models by identifying and extracting patterns from the datasets of test inputs and their outputs obtained in Con2.1.

Evaluation We empirically evaluate the inferred failure models in terms of accuracy, precision, and recall and study how effectively they capture and explain system failures.

RQ3: How can we identify unrealistic and invalid inputs for CPS? In this research question, we aim to identify unrealistic or invalid inputs whose outcomes are spurious because they violate requirement preconditions, lie outside the ODD, or correspond to safe/low-risk situations. Our contributions within the scope of RQ3 are:

Con3.1 We propose ML- and GP–based approaches that infer test-input validators as sets of conditions characterizing scenarios that violate requirement preconditions or ODD limits, as well as safe or low-risk scenarios. Our GP algorithm uses well-known spectrum-based fault localization (SBFL) ranking formulas, namely Tarantula [JH05], Ochiai [AZVG07] and Naish [NLR11], as fitness functions.

Con3.2 We provide a formal characterization of the expressive power of the inferred test validators in capturing common CPS signal properties.

Evaluation We assess the accuracy and robustness of the inferred test-input validators. Further, using a human-subject study, we measure the alignment of the inferred test validators as the degree to which the learned assertions align with the descriptions of precondition violations, ODD limits, and low-risk scenarios provided in technical standards as well as in empirical and expert-validated studies [HJTM18, CAK25, ITU25, Pra21, YSYA19, CWX24, LSWH24, WAYZ25, WAWZ25, BNPR20, BES+23, ANBS18, aut25, Adm09].

RQ4: Can we accurately capture conditions under which CPS exhibits non-robust behaviours? In this research question, we aim to find conditions under which the system exhibits non-robust behaviours, i.e., behaviours where a small change in the input shifts the output from passing to failing or vice versa. Our contribution within RQ4 is:

Con4.1 We propose a data-driven approach to identify conditions under which the system demonstrates non-robust behaviours. Specifically, we leverage ML models to guide

the sampling process toward regions that are more likely to exhibit such behaviours and to infer the conditions under which these behaviours occur.

Evaluation We evaluate the inferred conditions using precision and recall in predicting non-robust system behaviours and obtain feedback from a domain expert on their usefulness.

Contributions Con1.1, Con2.1, and Con2.2 were published as a journal article [JCNS24] in ACM Transactions on Software Engineering and Methodology (TOSEM) and are presented in Chapter 4. Contribution Con3.1 is a journal article [JGSN25] currently under review at IEEE Transactions on Software Engineering (TSE) and is presented in Chapter 6. Contribution 4.1 was published as a conference paper [JNSS23] at the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2023) and is presented in Chapter 5.

1.4.2 Research Papers

The core contributions presented in this thesis are based on the publications listed below:

- Jodat, B. A., Nejati, S., Sabetzadeh, M., & Saavedra, P. (2023). Learning non-robustness using simulation-based testing: A network traffic-shaping case study. In 16th IEEE Conference on Software Testing, Verification and Validation (ICST 2023) (pp. 386-397). IEEE.

- Jodat, B. A., Chandar, A., Nejati, S., & Sabetzadeh, M. (2024). Test Generation Strategies for Building Failure Models and Explaining Spurious Failures. ACM Transactions on Software Engineering and Methodology (TOSEM), 33(4), 1-32.

- Jodat, B. A. (2024). Insights into System Failures: ML-Assisted Testing and Failure Models for Cyber-Physical Systems. In 17th IEEE Conference on Software Testing, Verification and Validation (ICST 2024) (pp. 460-462). IEEE.

- Jodat, B. A., Gaaloul, K., Sabetzadeh, M., & Nejati, S. (2025). Automated Test Validators for Flaky Cyber-Physical System Simulators: Approach and Evaluation. *Under review* in IEEE Transactions on Software Engineering (TSE). IEEE.

1.5 Thesis Structure

This thesis is divided into eight chapters and is organized as follows:

Chapter 1 motivates effective and reliable testing of CPS, presents the research questions, and outlines the contributions.

Chapter 2 provides fundamental background on supervised ML techniques, and meta-heuristic search algorithms.

Chapter 3 presents our case-study systems, which are from networking, autonomous driving, and industrial-scale Simulink models.

The subsequent three chapters each focus on different challenges of effective and reliable testing of CPS and are based on one of my prior publications. Considering that these

chapters are based on multi-author publications, I include below a summary of my own contributions to the corresponding publications where relevant. Unless otherwise noted, I am the sole contributor to the thesis.

Chapter 4 presents our approaches for efficient test generation and building failure models to explain the reasons behind system failures for CPS.

- This chapter is based on my publication [JCNS24]. My contributions include formulating the core research questions, and conceptually designing the framework and methodological flow for efficient test generation and failure modelling. I led the design and technical implementation of the approaches for our case-study systems. Further, I contributed to the evaluation design and comparing it with state-of-the-art and baselines. One of the co-authors helped with implementing approaches for Simulink models and assisted in running some of the repetitive evaluation experiments. Co-authors helped by providing initial research ideas and offering continuous and in-depth feedback during the writing and review process.

Chapter 5 introduces our simulator for a network traffic shaping system, as well as our approach to learning non-robust behaviours of the network system.

- This chapter is based on my publication [JNSS23]. My contributions include identifying the research gaps, formulating the problem, and conceptually designing the framework for simulation and non-robust behaviour learning. Further, I developed the simulator, implemented our approach for learning non-robust behaviours, and designed the evaluation and baseline comparison. Co-authors contributed with continuous, in-depth feedback during writing.

Chapter 6 presents our approach for inferring assertion-based test validators for CPS.

- This chapter is based on my publication [JGSN25], which is currently under review. My contributions include identifying challenges in automated test validation for CPS, formulating the research problem, and conceptually designing the framework and methods for assertion inference. In addition, I led the technical implementation of the approaches, designed the evaluation strategy, led the human-subject study, and analyzed the results. Co-authors contributed initial research ideas and provided continuous, in-depth feedback during the writing and review process.

Chapter 7 elaborates on the related work pertaining to testing CPS.

Chapter 8 concludes the thesis with a summary and an outline of our future research directions.

Chapter 2

Background

This chapter outlines the foundational concepts explored in this thesis. Section 2.1 covers key concepts in supervised ML and provides background on the ML models used in this thesis, focusing on classification and regression tasks. Section 2.2 presents metaheuristic search algorithms, with emphasis on random search and genetic programming. Section 2.3 introduces spectrum-based fault localization and the suspiciousness formulas used in this thesis. Finally, Section 2.4 summarizes this chapter.

2.1 Supervised Machine Learning

Supervised ML is a type of learning paradigm in which a model is trained using labelled datasets that contain both input features and known output labels. The goal is for the model to learn a mapping from inputs to outputs so that it can predict the label of unseen instances. This learning process involves minimizing error between predicted outputs and

true labels by adjusting internal model parameters. Supervised learning problems are typically divided into two main categories: (1) classification and (2) regression, depending on the nature of the output variable.

(1) Classification is concerned with predicting a discrete label or category for an input instance. Given a training dataset with examples belonging to known classes, a classification algorithm learns patterns and decision boundaries that can assign new inputs to one of the predefined categories. Common algorithms used in classification tasks include decision trees, decision rules, support vector machines, k-nearest neighbors, and neural networks.

(2) Regression involves predicting a continuous numeric value rather than a discrete label. It is used in scenarios where the output variable can take on any real value, such as predicting housing prices [MAP19], water temperature [QHXJ22], or stock market trends [ARKA19].

In addition to the distinction between classification and regression, supervised ML models can also be categorized as **interpretable** or **non-interpretable** [Mol20]. Interpretable models, such as decision trees, linear regression, or rule-based classifiers, provide clear reasoning behind their predictions, making it possible for humans to understand how different inputs affect outputs. These models are often favoured in domains where transparency is critical, such as autonomous driving and healthcare systems. Non-interpretable models, including deep neural networks, ensemble methods like random forests and gradient boosting, or support vector machines, offer limited insight into their internal decision-making processes. While such models excel in handling high-dimensional data and capturing com-

plex relationships, their lack of transparency poses challenges in domains that require explainability or regulatory compliance.

In this thesis, we employ both interpretable and non-interpretable ML models. We first detail the interpretable models: decision trees, decision rules, and logistic regression. We then explain the non-interpretable models: Gaussian process regression, gradient boosting, neural networks, random forest, and support vector regression.

2.1.1 Interpretable ML Models

In this section, we explain three interpretable ML models used in this thesis: (1) decision trees, (2) decision rules, and (3) logistic regression.

(1) Decision Trees. A decision tree is a hierarchical model that represents decisions and their possible outcomes in a tree-like structure composed of nodes and branches. It is typically constructed using greedy recursive algorithms such as CART [BFOS84] or ID3/C4.5 [Qui93]. During construction, the algorithm recursively splits the input space into regions by applying tests on feature values, with the goal of producing regions that are as homogeneous as possible with respect to the target variable.

A decision tree begins with a root node at the top, which represents the first and most informative decision point. Internal nodes capture subsequent decisions by applying thresholds to feature values, while branches indicate the outcomes of these tests and connect nodes together. The process ends at leaf nodes, which provide the final prediction – either a class label in classification tasks or a numerical value in regression tasks.

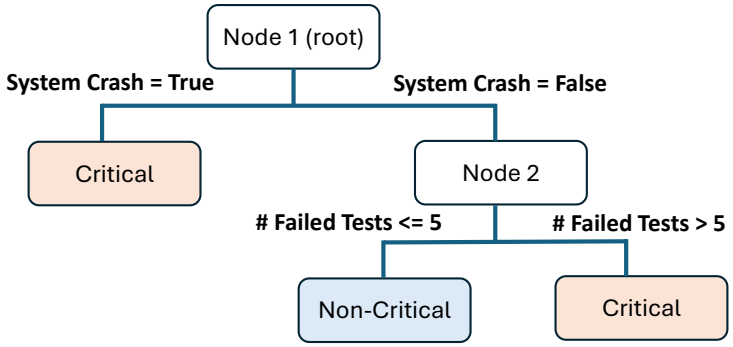


Figure 2.1: An example of a decision tree for classifying critical or non-critical bugs.

An example of a decision tree is shown in Figure 2.1. This tree classifies bugs as critical or non-critical based on whether a system crash has occurred and the number of failing tests. For instance, a bug is classified as non-critical if no system crash has occurred and the number of failing tests is less than five.

(2) Decision rules. Decision rules are a set of human-readable logical statements derived from a decision tree or learned directly by rule-based classifiers such as RIPPER (Repeated Incremental Pruning to Produce Error Reduction) [Coh95]. Each rule takes the form of an IF-THEN statement, specifying the conditions under which a certain prediction is made. For example, when deciding whether to play tennis, a rule can be: IF Temperature > 75 AND Humidity ≤ 80 THEN PlayTennis = Yes.

Rule-based classifiers like RIPPER construct decision rules directly by iteratively adding conditions that maximize accuracy on the training set while pruning to avoid overfitting. This pruning ensures that the rules generalize well to unseen data.

(3) Logistic regression. A logistic regression model estimates the probability that an instance belongs to a given class by applying a logistic (sigmoid) function to a linear combination of the input features. Formally, the model is defined as [log25, WFH11]:

$$P(Y = 1 \mid X) = \frac{1}{1+e^{-(c_0+c_1x_1+c_2x_2+\cdots+c_nx_n)}}$$

where:

- Y is the binary outcome (dependent variable), coded as 1 for the positive class (e.g., a critical bug) and 0 for the negative class (e.g., a non-critical bug).
- $X = (x_1, \ldots, x_n)$ is the vector of input features (independent variables), each representing a measurable property of the instance (e.g., whether a crash occurred, number of failed tests).
- $P(Y = 1 \mid X)$ is the predicted probability that the outcome is positive given the features.
- c_0 is the model's intercept, and $c_1, \ldots, c_n$ are the coefficients, each representing the weight of its corresponding feature x_i.

Logistic regression provides coefficients $(c_1, \ldots, c_n)$ that can be directly assessed to understand the influence of each feature on the predicted outcome. For example, when predicting whether a bug will be critical, a positive coefficient for crash occurrence indicates that the presence of a crash strongly increases the likelihood of the bug being classified as critical. Because these feature coefficients explicitly quantify the direction and strength of

influence, they enable engineers to make transparent, evidence-based decisions, making logistic regression particularly valuable in domains where explainability and accountability are essential.

2.1.2 Non-Interpretable ML Models

In this section, we explain non-interpretable ML models used in this thesis: (1) Gaussian process regression, (2) gradient boosting, (3) neural networks, (4) random forest, and (5) support vector regression.

(1) Gaussian process regression. Gaussian process regression (GPR) is a powerful, non-parametric Bayesian technique for regression. Unlike parametric models like linear regression or neural networks that assume a specific functional form for the data, GPR instead models a probability distribution over all possible functions that could fit the training data. This makes GPR particularly well-suited for problems where flexibility and principled uncertainty estimates are essential.

GPR defines a prior distribution over all possible functions before any training data are observed. The prior distribution is specified by a mean function, often taken to be zero, and a covariance (kernel) function, which encodes assumptions about the properties of the target function. The kernel plays a central role in GPR because it determines the similarity between data points, thereby shaping the smoothness, continuity, or other characteristics of the functions being modelled. By choosing different kernels, engineers can encode domain knowledge about the underlying functions – for example, smoothness

using the radial basis function (RBF) kernel, periodicity for time-series data, or linear trends for structured prediction problems.

When training data are observed, Bayesian inference updates the prior distribution into a posterior distribution over functions. This posterior refines the set of likely functions by incorporating the observed data, focusing on functions that closely align with the training points. The resulting posterior defines the predictive distribution for unseen inputs, consisting of a mean (the regression prediction) and a variance (the model's uncertainty).

(2) Gradient boosting. Gradient boosting is a widely used ensemble learning technique that constructs a strong predictive model by sequentially combining multiple weaker models, most often decision trees. Rather than attempting to learn a complex mapping from inputs to outputs in a single step, gradient boosting follows an iterative process in which models are added one at a time to gradually improve predictive performance. Each new model in the sequence is trained to correct the errors of the ensemble built so far, thereby incrementally refining the overall accuracy of the system.

The central idea behind gradient boosting is to frame the learning task as an optimization problem. The algorithm begins with a simple base model, such as a shallow decision tree, which provides initial predictions. The difference between these predictions and the true values, often referred to as the residuals, indicates where the model has performed poorly. Instead of directly fitting the next model to the target values, gradient boosting fits it to these residuals, effectively training the model to predict and correct errors. This correction process is guided by the gradient of a chosen loss function, which ensures that each successive model reduces the overall error in the direction of steepest descent.

Over multiple iterations, the ensemble accumulates a series of models, each focused on addressing the shortcomings of its predecessors. Although the individual models are weak learners on their own, their collective contribution produces a highly accurate predictor. Gradient boosting is particularly powerful because it can adapt to a wide range of loss functions, making it suitable for both regression and classification tasks. Moreover, by incorporating techniques such as learning rate adjustment, regularization, and tree depth constraints, gradient boosting can achieve strong generalization while mitigating the risk of overfitting [IBM25].

(3) Neural networks. Neural networks are a class of computational models inspired by the structure and functioning of the human brain. At a high level, they consist of interconnected layers of nodes, commonly referred to as neurons, which process information and transmit signals to one another. Each connection between neurons carries a weight, representing the strength of influence that one neuron has on another. By adjusting these weights during training, neural networks are able to learn complex patterns in data.

The architecture of a neural network typically includes an input layer, one or more hidden layers, and an output layer. The input layer receives raw data, while the hidden layers perform transformations through a series of weighted computations and non-linear activation functions. The output layer generates predictions or classifications based on the processed information. This layered structure enables neural networks to model highly non-linear relationships. An example of a neural network with three layers is in Figure 2.2. In this example, the network has one input layer with three neurons, one hidden layer with four neurons, and one output layer with one neuron.

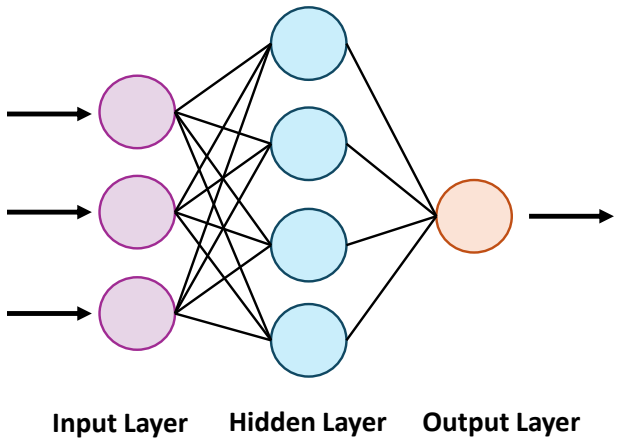


Figure 2.2: An example of a neural network with an input layer (3 neurons), a hidden layer (4 neurons), and an output layer (1 neuron).

Learning in neural networks is generally achieved through a process known as back-propagation, combined with optimization techniques like gradient descent. During training, the network compares its predictions with the actual outcomes, calculates an error, and propagates this error backward through the network to update the weights. Over many iterations, the network reduces the error and improves its performance on the given task.

(4) Random forest. Random forest is an ensemble learning method that extends the principles of decision trees to achieve greater predictive accuracy. Random forest method constructs a large number of decision trees during training, where each tree is built on a random subset of the training data and a random subset of the features. This use of randomness – often referred to as bootstrap aggregation (bagging) of samples and feature randomness – ensures diversity among the trees, thereby reducing the tendency of a single decision tree to overfit the training data.

When making predictions, each tree in the forest produces its own output, and the final decision is obtained by aggregating these outputs. For classification tasks, random forest applies majority voting, selecting the class predicted by most of the trees. For regression tasks, random forest computes the average of the predictions across trees. This aggregation process allows random forest to combine the strengths of many weak learners (individual decision trees), yielding a model that is more accurate and generalizable than any single tree trained on the data.

Beyond predictive accuracy, random forest also provides insights into the role of different input variables in the learning process. During training, it can generate measures of feature importance, indicating which input variables contribute most to the model's decisions. Its ensemble nature also makes it robust to noise and outliers within the training data.

(5) Support vector machine and regression. Support vector machines (SVM) aim to find an optimal hyperplane that separates data points belonging to different classes with the largest possible margin. The principle of maximizing the margin ensures that the classifier achieves good generalization, as it seeks a decision boundary that is both simple and robust. When the data is not linearly separable in its original space, SVMs employ kernel functions to project the input data into a higher-dimensional feature space where a linear separation becomes feasible. This kernel approach allows SVMs to capture complex, non-linear relationships without explicitly computing the high-dimensional mapping.

Support vector regression (SVR) adapts the SVM framework to handle regression tasks where the goal is to predict continuous values rather than discrete class labels. Instead of finding a hyperplane that separates classes, SVR attempts to find a regression function

that approximates the data while maintaining a balance between accuracy and complexity. The method introduces the concept of an ϵ-insensitive region, which specifies a margin of tolerance around the regression function. Errors that fall within this ϵ-region are disregarded, while errors outside it are penalized. This design ensures that SVR focuses only on significant deviations.

Similar to SVMs, the regression function in SVR is determined primarily by a subset of the training examples known as support vectors. These are the examples that lie on or outside the ϵ-insensitive region and are critical in shaping the regression function. By relying only on these influential training examples, SVR achieves a sparse representation, which improves computational efficiency. Furthermore, through kernel functions such as the radial basis function (RBF), SVR is able to model highly non-linear relationships between predictors and outcomes.

2.2 Metaheuristic Search

In this section, we introduce metaheuristic search and explain two algorithms used in this thesis: random search and genetic programming.

Metaheuristic search techniques are high-level procedures designed to find near-optimal solutions for complex optimization problems where exact methods, such as exhaustive search, are computationally infeasible. These techniques provide a stochastic approach to exploring large and poorly understood search spaces [Luk13]. By incorporating adaptive

strategies, metaheuristics can escape local optima, making them highly effective for tackling real-world challenges, such as optimization tasks in engineering and software testing.

Metaheuristic algorithms can be broadly classified into two categories: single-state methods (e.g., hill climbing and simulated annealing) and population-based methods (e.g., genetic algorithms and particle swarm optimization). While single-state methods explore the search space using a single candidate solution, population-based methods evolve a population of solutions over time. In this section, we first explain a single-state method, random search, and then describe a population-based approach, genetic programming.

2.2.1 Random Search

Random search is a simple yet effective metaheuristic approach that explores the solution space by sampling candidate solutions uniformly at random. Unlike deterministic or gradient-based optimization methods, random search does not require gradient information or problem-specific heuristics.

Random search repeatedly generates new solutions by randomly selecting values for each input variable within predefined bounds. Each solution is evaluated using an objective or fitness function, and the best-performing solution is retained. This process is repeated until a satisfactory solution is found or a stopping condition is met (e.g., time or number of evaluations). Despite its simplicity, random search can perform well in high-dimensional spaces where other methods may get stuck in local optima. However, its major limitation is the lack of exploitation: it does not use any information about previously evaluated solutions to guide future search, which can lead to inefficiencies and slow convergence.

Random search often serves as a baseline for comparing the performance of more sophisticated metaheuristics.

2.2.2 Genetic Programming

Genetic programming (GP) is an evolutionary search technique that extends genetic algorithms to evolve executable computer programs or symbolic expressions. Inspired by the principles of natural selection and genetics, GP maintains a population of candidate solutions represented as tree-like structures, with internal nodes denoting functions and leaf nodes denoting terminals such as input variables or constants.

Algorithm 2.1 shows the standard GP workflow. The algorithm starts by generating an initial population and evaluating each individual using a fitness function (Lines 1–3). It then iteratively breeds offspring via crossover and mutation and forms the next generation through survivor selection (Lines 4–7). This evolutionary cycle continues until a stopping condition is satisfied (Lines 4–9). The algorithm finally returns the best evolved solution(s) (Lines 10–11). The typical workflow of GP can be summarized as follows:

1. Initialization. Generate an initial population of random program trees composed of functions and terminals relevant to the problem domain.

2. Fitness Evaluation. Evaluate each program based on how well it performs the target task using a fitness function.

3. Selection. Select high-performing programs (parents) using methods such as tournament selection or roulette wheel selection.

Algorithm 2.1 Genetic Programming Workflow

Input F: Fitness function
Output S: Set of best-evolved solutions

```
t ← 0;
P_0 ← Initialize(); //Generate an initial population
P_0.Fit ← Evaluate(P_0, F); //Compute fitness of individuals
while ¬(stop condition) do
  off ← Breed(P_t); //Apply crossover and mutation
  off.Fit ← Evaluate(off, F);
  P_{t+1} ← select individuals from P_t ∪ off;
  t ← t + 1;
end
S ← BestSolutions(P_t);
return S
```

4. Genetic Operations. Create offspring by applying genetic operators to selected parents using

   - Crossover: Swap sub-trees between two parent programs to create offspring.

   - Mutation: Replace a randomly selected sub-tree with a newly generated random sub-tree.

   - Reproduction: Optionally copy top individuals directly to the next generation (elitism).

5. Termination Check. Repeat steps 2–4 for a fixed number of generations or until a stopping condition is met (e.g., fitness threshold or stagnation).

6. Output the Best Program. Return the best-performing individual(s) as the final evolved solution.

To effectively apply GP, several key design choices must be considered. First, the selection of the function set (e.g., arithmetic operators, logical functions) and terminal set (e.g.,

input variables, constants) must be appropriate for the problem domain to ensure that evolved programs can express potential solutions. Second, the representation of individuals – commonly as syntax trees – must support syntactic validity after genetic operations. Third, the design of the fitness function is crucial, as it guides the evolutionary process by quantifying the quality of candidate programs. Poorly designed fitness functions may lead to premature convergence or the evolution of programs that overfit training data. Fourth, the balance between crossover and mutation must be carefully tuned. Crossover promotes exploitation by combining useful substructures from parents, while mutation encourages exploration by introducing novel structures. Over-reliance on crossover may reduce diversity and lead to premature convergence, whereas excessive mutation may disrupt useful patterns and hinder progress. A well-balanced strategy ensures both effective refinement of solutions and continued exploration of the search space.

GP's flexibility and evolutionary search make it well suited to tackling test-input validity in CPS (Challenge 3 in Section 1.2). By evolving human-readable constraints over input signals, GP can produce test-input validators that are accurate while remaining easy for developers and domain experts to understand and apply.

2.3 Spectrum-based Fault Localization

Fault localization aims to help engineers identify parts of a system that are likely responsible for failures. Classical fault localization approaches rely on program analysis and debugging techniques that often require substantial manual effort. In contrast, spectrum-based fault localization (SBFL) uses information about which program elements are exercised by

which tests, together with pass or fail outcomes, to compute a suspiciousness ranking of elements [JH05, AZVG07, NLR11].

SBFL relies on the notion of an *execution spectrum*, which summarizes how tests exercise system elements. Consider a set of tests (TS), each labelled as passing or failing, and a set of elements whose involvement in each test execution can be recorded. In software fault localization, elements are statements, branches, or predicates in the program under test. For each element, SBFL collects four counts:

- c_f: the number of failing tests that *execute* the element,
- c_p: the number of passing tests that *execute* the element,
- c_{nf}: the number of failing tests that *do not execute* the element,
- c_{np}: the number of passing tests that *do not execute* the element.

These counts form a contingency table that captures the association between execution of the element and failure of the tests. SBFL formulas then map $(c_f, c_p, c_{nf}, c_{np})$ to a real-valued suspiciousness score. The higher the score, the stronger the statistical association between exercising the element and observing failures.

Let TS_p be the set of passing tests and TS_f the set of failing tests, and let E be the set of elements. For an element $e \in E$ and test t, a predicate $exec(e, t)$ indicates whether e is executed when running t. The counts are defined as:

$$c_f(e) = \left|t \in TS_f \mid exec(e,t)\right|,$$

$$c_p(e) = \left|t \in TS_p \mid exec(e,t)\right|,$$

$$c_{nf}(e) = \left|t \in TS_f \mid \neg exec(e,t)\right|,$$

$$c_{np}(e) = \left|t \in TS_p \mid \neg exec(e,t)\right|.$$

A suspiciousness function *susp* assigns a real number to each element e based on these counts:

$$susp(e) = f\big(c_f(e), c_p(e), c_{nf}(e), c_{np}(e)\big),$$

where f is chosen from a family of formulas proposed in the literature. Different formulas encode different intuitions about how strongly execution of an element should correlate with failures in order to be considered suspicious.

Many suspiciousness formulas have been proposed and evaluated empirically [WGL+16]. The following sections introduce the three formulas used in this thesis, namely Tarantula, Ochiai, and Naish, which originate from statistical and similarity-based reasoning.

2.3.1 Tarantula

The Tarantula formula [JH05] is motivated by the intuition that an element is more suspicious when it is executed frequently by failing tests and relatively rarely by passing tests. Let $|TS_f|$ denote the total number of failing tests and $|TS_p|$ the total number of passing tests. The Tarantula suspiciousness of an element e is defined as:

$$Tarantula(c) = \frac{\frac{c_f(c)}{|TS_f|}}{\frac{c_p(c)}{|TS_p|} + \frac{c_f(c)}{|TS_f|}}.$$

The numerator captures the proportion of failing tests that execute e, and the denominator normalizes this quantity by the total proportion of tests, both failing and passing, that execute e. An element that appears predominantly in failing tests obtains a score close to 1, while an element covered mostly by passing tests obtains a score close to 0.

2.3.2 Ochiai

The Ochiai formula [AZVG07] is inspired by a similarity coefficient used in biology and information retrieval. It measures how strongly the execution of an element correlates with failure, relative to the overall frequency of both failures and executions:

$$Ochiai(c) = \frac{c_f(c)}{\sqrt{|TS_f| \cdot (c_p(c) + c_f(c))}}.$$

The numerator counts failing executions of e, while the denominator scales this count by the geometric mean of the total number of failing tests and the total number of tests that execute e. Elements that are executed in many failing tests and few passing tests receive higher scores.

2.3.3 Naish

The Naish formula [NLR11] prioritizes elements that are executed by many failing tests and by no passing tests, while penalizing elements that frequently appear in passing executions:

$$Naish(c) = \frac{c_f(c)}{|TS_f|} - \frac{c_p(c)}{1 + |TS_p|}.$$

If an element is never executed by failing tests, its suspiciousness becomes negative, placing it at the bottom of any ranking. If it is executed by some failing tests, the score grows with c_f and is reduced when the element is executed by many passing tests. This formula encodes a strong preference for elements that appear only in failing executions.

In this thesis, Tarantula, Ochiai, and Naish are used as fitness functions of GP to assess how strongly certain predicates over system inputs associate with passing and failing behaviours of the SUT. Specifically,

- Candidate predicates act as *elements* in the SBFL sense.
- Executions that satisfy a predicate correspond to tests that "execute" the element, while executions that do not satisfy it correspond to tests that do not execute the element.

GP then maximizes this score to evolve predicates that are either strongly associated with failures or strongly associated with passes.

2.4 Summary

This chapter presents the necessary background to support the understanding of the core concepts explored throughout this thesis. Section 2.1 presented the foundations of supervised ML, distinguishing between classification and regression tasks, and explained both interpretable (e.g., decision trees, decision rules, logistic regression) and non-interpretable models (e.g., Gaussian process regression, gradient boosting, neural networks, random

forest, and support vector regression). Section 2.2 provided an overview of metaheuristic search algorithms, focusing on random search and genetic programming. Section 2.3 described the principles of SBFL and the Tarantula, Ochiai, and Naish formulas.

Chapter 3

Case-Study Systems

In this chapter, we provide detailed information on the case-study systems we use in this thesis to evaluate our approaches. These case-study systems span domains including industrial-scale Simulink models (Section 3.1), autonomous driving (Section 3.2), and networking systems (Section 3.3). For each domain, we evaluate on an executable environment, either a simulator or a controlled test bed, that lets us run the system repeatedly under configurable inputs and observe its outputs. All inputs for our case-study systems are numeric, though they vary in measurement and scale. In industrial-scale Simulink models, these numeric inputs include time-varying signals for model inputs (e.g., control signals, sensor readings) and parameters used to simulate and analyze system dynamics. For autonomous driving systems, inputs are numeric sensor readings from LiDAR, radar, and other sensors, which are processed to navigate the vehicle. In networking systems, inputs represent numeric measurements of bandwidth for data flows going through routers. By employing this diverse set of case-study systems, our evaluation captures a wide range

of real-world systems, demonstrating that our approaches are robust and adaptable across domains with different characteristics.

3.1 Simulink Models

Simulink [Cha17] is a powerful MATLAB toolbox extensively used for the specification, simulation, and analysis of CPS. Together, MATLAB and Simulink provide engineers with a versatile environment for designing and developing advanced, industry-specific solutions across domains such as automotive, aerospace, telecommunications, and more. These tools support diverse engineering needs by enabling in-depth system modelling in areas of signal and image processing, computer vision, communications, control design, and robotics [Gaa21].

Simulink's extensive library of pre-built blocks, input/output ports, and connections offers an intuitive approach to modelling complex systems. In this library, blocks typically represent operations or constants within the system, while ports define any data or signals exchanged between blocks. Connections illustrate data flow, allowing for cohesive interaction throughout the model [Gaa21]. This modular structure enables engineers to efficiently prototype, test, and refine system designs. Figure 3.1 presents a Simulink model example of a De Havilland Beaver airframe integrated with an autopilot system.

The inputs and outputs of a Simulink model are represented using signals. Figure 3.2(a) shows an example test input with seven signals for the autopilot Simulink model in Figure 3.1, and Figure 3.2(b) shows the corresponding output with two signals.

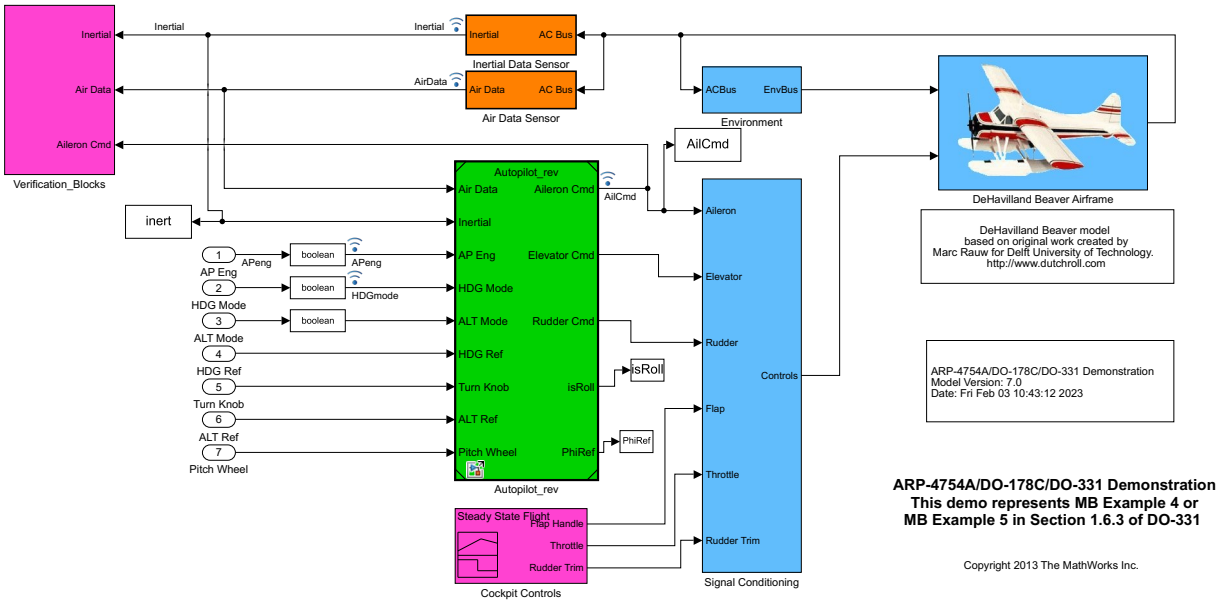
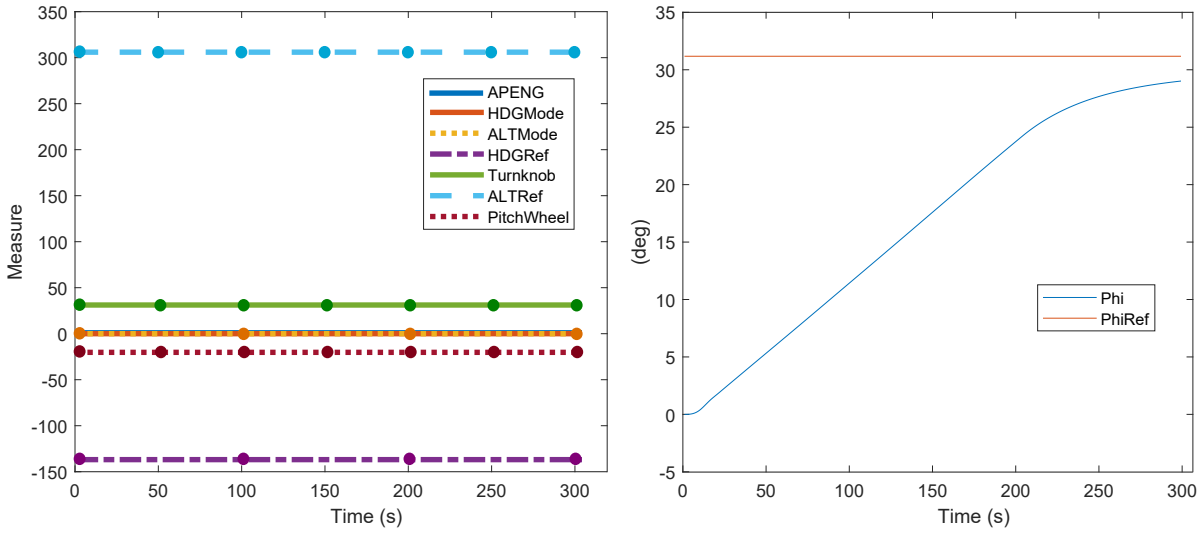


Figure 3.1: An example of a Simulink model

Let M be a Simulink model with input signals. We denote each test input for M as $\bar{u} = \{u_1, u_2, \ldots, u_m\}$ where each u_i is a signal for an input of M over a time domain $\mathbb{T} = [0, b]$, i.e., $u_i : \mathbb{T} \rightarrow \mathbb{R}$. For example, the autopilot Simulink model in Figure 3.2 shows the values of seven input signals over the time domain $\mathbb{T} = [0, 300s]$. A typical input-signal generator for Simulink uses *control-point encoding* [TFP+19, AWM+19] – a common approach in signal processing and control theory that captures signals in a compact and efficient representation. Specifically, in this approach, signals are encoded as sequences of equally spaced control points, where each index denotes a fixed time interval and each corresponding value approximates the signal's value during that time interval. Following this encoding, to generate signals, it suffices to generate the control points of the signal. Provided with the control points, the actual signals are constructed through interpolation. An interpolation function (e.g., piecewise constant, linear, or cubic) connects the control



(a) Input Signals (b) Output Signals

Figure 3.2: An example of input and output signals for the autopilot simulink model shown in Figure 3.1

points to form a signal [TFP+19, AWM+19, GMN+20]. In this thesis, we assume the interpolation function for our Simulink system inputs is piecewise constant. Our evaluation of major CPS benchmarks and case studies from the literature indicates that most assume inputs to be piecewise constant. Notably, the Lockheed Martin industrial CPS benchmark and the MathWorks' publicly available CPS models for automotive driving systems – including the cruise controller [cru25], clutch lockup controller [clu25], guidance control system [gui25], and DC motor controller [dcm25] – explicitly use piecewise-constant input signals. In addition, the ARCH benchmark [KFA+24], which includes seven hybrid CPS models, compares two interpolation function types: arbitrary piecewise continuous functions and piecewise constant functions. Results from ARCH-COMP19 [EAD+19] show that

performance differences between these two interpolation functions are minimal, indicating that our assumption of piecewise-constant signals is not restrictive.

Let $u : \mathbb{T} \rightarrow \mathbb{R}$ be an input signal. We encode u using n_u control points, i.e., $c_{u,0}$, $c_{u,1}$, $\ldots$, c_{u,n_u-1}, equally distributed over the time domain $\mathbb{T} = [0, b]$, i.e., positioned at a fixed time distance $I = \frac{b}{n_u-1}$. Let $c_{x,y}$ be a control point, x is the signal the control point refers to, and y is the position of the control point. The control points $c_{u,0}$, $c_{u,1}$, $\ldots$, c_{u,n_u-1} respectively contain the values of the signal u at time instants $0, I, 2 \cdot I, \ldots, (n_u - 1) \cdot I$. For example, in Figure 3.2, signals ALTRef, Turnknob and Pitchwheel are represented using seven control points, while signals APEng, HDGMode, ALTMode and HDGRef are represented using four control points over the $[0, 300s]$ time domain.

CPS are expected to satisfy a number of system-level requirements that are critical to their correct and robust operation in real-world environments. These requirements, which emerge from the deep integration of computational algorithms with physical processes, typically encompass safety, security and reliability.

To determine the degree to which test inputs satisfy or violate system requirements, mathematical functions, called fitness functions, are formulated to quantitatively evaluate test outcomes with respect to system requirements. These functions assign a numerical value that reflects how well a given test input exposes desired behaviours or uncovers deviations from expected system performance. For each requirement, we define a fitness function by encoding it in Restricted Signals First-Order Logic (RFOL), a variant of signal temporal logic that effectively captures many requirements of CPS [MNGB19]. Our fitness functions utilize the semantic function of RFOL to quantify test outcomes.

Fitness functions distinguish among passing tests (those that meet the requirement) and failing tests (those that violate the requirement). Specifically, we assume the range of the fitness function F is an interval $[-a, b] \in \mathbb{R}$. For a given test, $F(t) \geq 0$ iff t is passing, and otherwise, t is failing. The closer $F(t)$ is to b, the higher the confidence that t passes; and the closer $F(t)$ is to $-a$, the higher the confidence that t fails. Based on these fitness functions, a pass/fail verdict can be systematically derived for any test input.

In this thesis, we use an industrial-scale and public-domain benchmark of Simulink specifications from Lockheed Martin [loc25]. This benchmark provides a set of representative CPS systems that are shared by Lockheed as verification-and-validation challenge subjects for researchers and quality-assurance tool vendors. The benchmark is comprised of eleven specifications. Table 3.1 shows a list of these specifications along with their descriptions, number of blocks and number of requirements.

Throughout our experiments with the Lockheed Martin benchmark, we ensure realism by generating inputs that reflect the physical and operational characteristics of the systems being modelled. This includes avoiding abrupt, contradictory or physically implausible signal changes and instead generating input trajectories that evolve smoothly, remain within feasible ranges and respect correlations between related signals. For example, in the finite state machine, we prevent instantaneous on/off flips of signals like standby (pilot-in-control) and apfail (external failure indication), which would not occur in practice. Similarly, for the two-tank model, we avoid instantaneous valve commands, generating instead gradual open/close profiles that match the constraints of physical actuation. For the autopilot model, we ensure realism by avoiding abrupt or conflicting changes in signals such as throttle and pitch angle that would violate aircraft dynamics.

Table 3.1: Names, descriptions, the number of blocks and the number of requirements of our benchmark Simulink models.

Name	Description	#Blocks	#Reqs
Tustin	A common flight control utility for computing the Tustin Integration	57	9
Regulator	A regulators inner loop architecture used in many feedback control applications	308	1
Nonlinear Guidance	A nonlinear algorithm for generating a guidance command for an aircraft	373	1
Finite State Machine	A finite state machine to enable autopilot mode if a hazardous situation is identified	303	1
Neural Network	A neural network model with a two-input, single-output structure, featuring two hidden layers in a feed-forward architecture	704	3
System Wide Integrity Monitor	A numerical algorithm that alerts the operator when the airspeed is nearing a threshold beyond which an evasive fly-up maneuver cannot be performed	164	2
Effector Blender	A control allocation method that calculates the optimal configuration of effectors for a vehicle	95	3
Two Tanks	A two-tank system that controls liquid flow using sensors and valves to maintain safe levels, with emergency drainage to prevent overflow.	498	32
Euler	A mathematical model that generates three-dimensional rotation matrices for the z-, y-, and x-axes of an inertial frame in Euclidean space.	834	8
Triplex	A monitoring system that assesses readings from three redundant sensors to determine the trusted values for a safety-critical system.	481	4
Autopilot	A single-engine, high-wing, propeller-driven aircraft simulation with all six degrees of freedom	1549	3

3.2 Autonomous Driving System

Simulators for autonomous driving systems (ADS) have become an essential tool in the development, testing, and validation of autonomous vehicles. These simulators enable engineers to create virtual environments that replicate real-world driving conditions, allowing the ADS to be tested in a controlled setting. There are two types of ADS architectures, each with its own approach to vehicle control and decision-making:

- **PID-based ADS:** PID controllers are classical control algorithms used in many ADS designs [PID]. PID controllers rely on a feedback loop where the control output is adjusted based on the proportional, integral, and derivative errors between the

desired and actual vehicle states. PID controllers are known for their simplicity and effectiveness in handling specific tasks, such as steering and speed control [PID].

- **DNN-based End-to-End ADS:** Deep neural network (DNN)-based end-to-end ADS use machine learning techniques to directly map sensor inputs (such as images from cameras and data from Lidar) to driving commands like steering, acceleration, and braking. This design allows the ADS to learn from large amounts of driving data and adapt to complex, dynamic driving scenarios [HSNB21].

The above ADS controllers are widely implemented and tested in ADS simulation environments, such as BeamNG [bea25]. In this thesis, we focus on the BeamNG simulator, a widely used open-source tool for ADS testing [bea25]. BeamNG includes a soft-body physics engine that supports deformation and more realistic crash dynamics. Other simulators, such as Udacity, Apollo, SVL, and DeepDrive, are not considered in this thesis because their active development has stopped or their release cycles are too long.

We use two types of self-driving controllers as our ADS systems, both executed and tested using the BeamNG simulator. The first system is the autopilot controller of BeamNG, a classical self-driving controller [bea25,SSK21]. The second is DAVE2 [BdTD+16], a DNN model trained for end-to-end self-driving. To test the autopilot controller, we developed two simulation environments in BeamNG: (1) a complex town map with multi-lane roads, other vehicles, and various static objects along the roads, and (2) a simpler environment with a two-lane road without other vehicles and static objects. We refer to the former environment as TWN and the latter environment as SNG. The simple environment (i.e., SNG) is developed by the CPS Testing Tool Competition track at the SBFT workshop [sbf25]. Fig-

(a) Complex town map with other vehicles and static objects

(b) Simple two-lane road without any vehicles or static objects

Figure 3.3: An example of two environments in BeamNG simulator

ure 3.3(a) shows an example of a TWN environment and Figure 3.3(b) shows an example of a SNG environment in the BeamNG simulator.

The inputs for testing an ADS define the layout information, ambient conditions and mobile objects within the driving scenario. For example, in the TWN environment, the inputs specify a route map, weather conditions, time of day, maximum speed for the ego vehicle and many other parameters. The test input is executed on the simulator, where the ADS is fed sensory information including data from cameras, Lidar, radar and other sensors. In response, the ADS generates throttle, braking, and steering commands to control the vehicle's movement. These commands are then directed back to the simulator which guide the vehicle through the environment. Table 3.2 shows the list of inputs for each environment. To ensure realism in our experiments, we enforce plausible vehicle positions, feasible target positions and avoid initial collisions caused by physically implausible starting positions or unsafe spacing between vehicles.

Table 3.2: A list of inputs for each environment in BeamNG simulator

Environment	Inputs
TWN	route map, weather, time of day, speed of ego vehicle and non-ego vehicles, the number of non-ego vehicles, initial position of the ego vehicle and non-ego vehicles, destinations of ego vehicle and non-ego vehicles, ego vehicle and non-ego vehicle types
SNG	road shape, weather, time of day, speed of ego vehicle, initial position of the ego vehicle, destination of ego vehicle, ego-vehicle type

An ADS test setup consists of an ADS controller, which can be either PID-based or DNN-based, along with a testing environment. In our case, the testing environments are TWN or SNG. To determine whether a test passes or fails in an ADS setup, we consider the following system-level requirements adopted from prior studies [ANN24, sbf25]:

- **R1**: *The ego car should remain within its lane, only deviating when intentionally changing lanes.*
- **R2**: *The ego car should always maintain a safety distance from other vehicles.*
- **R3**: *The ego car should always maintain a safety distance from the sidewalk and static objects.*
- **R4**: *The ego car should reach the specified destination within the maximum simulation time duration.*

For each requirement, we adopt fitness functions from existing literature that quantitatively assess the extent to which a test input violates or satisfies the corresponding requirement [ANN24, sbf25]. For each fitness function, we adopt a threshold from prior studies. Based on these thresholds we can assign a pass/fail verdict to each test input [ANN24, sbf25]. For example, the fitness function for R1 evaluates the car's lateral

position relative to the lane boundaries throughout the simulation. If the car crosses the lane markers without an intentional signal for a lane change, the fitness function indicates a violation. If the deviation exceeds the threshold (e.g., 80% of the car's width is outside the lane boundaries), the test would fail, indicating that the car did not adhere to the lane-keeping requirement [sbf25]. Conversely, if the car stays within the lane boundaries or only deviates when changing lanes intentionally, the test would pass.

3.3 Network Traffic Shaping System

In this section, we present the simulator that we developed in collaboration with RabbitRun Technologies to execute tests and simulate various network-usage scenarios in a generic SOHO setting. The SOHO market has been growing in importance in recent years, and has been crucial during the COVID-19 pandemic with a major part of the workforce needing to work from home. Typically, a network traffic-shaping system (NTSS) is used to support high-quality connectivity for SOHO, in particular for *real-time streaming* applications such as voice and video. Specifically, NTSS works by dividing the total network bandwidth into classes with different priorities. The higher-priority classes are typically used for transmitting time-sensitive, streaming voice and video. Traffic shaping is applied at the edge of the network and on routers to control the *outbound* traffic, i.e., the traffic travelling from equipment out onto the Internet.

Figure 3.4 shows the physical view of a SOHO setting where SOHO users are connected to a router via different types of devices (PCs, phones, tablets, etc.). Through an internet modem, the router sends data packets from the SOHO users to some external users' IPs

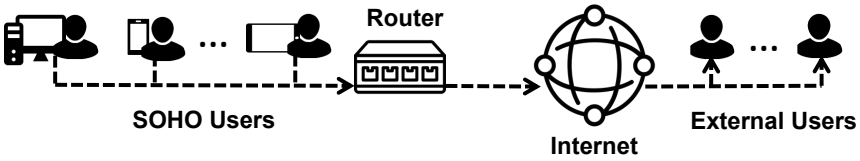


Figure 3.4: Overview of Small-Office/Home-Office (SOHO).

specified in the packets' headers. Note that an NTSS controls the outbound traffic only, i.e., the direction from SOHO users to the external users in Figure 3.4.

Our simulator, named *SOHOSim*, generates outbound traffic streams in the SOHO setup of Figure 3.4 and measures the quality of the network. Using virtual machines, SOHOSim captures all the elements in Figure 3.4, namely SOHO users, the router and external users. SOHOSim can simulate parallel data flows and streams (e.g., Zoom calls, VOIP, or gaming) sent by several simultaneous SOHO users. This enables us to run a large number of tests involving several parallel network flows sent by several users capturing many different realistic situations. Testing such situations on a physical setup is expensive and time-consuming. SOHOSim can measure network quality for SOHO users to, for example, determine if the users are experiencing network problems (e.g., choppy Zoom calls). In this way, the testing performed using SOHOSim allows us to determine whether an NTSS installed on the router is configured properly.

A conceptual model for SOHOSim is shown in Figure 3.5. For the purpose of testing an NTSS, our simulator needs to generate two types of flows: *data flows* and *control flows*. Each data flow has a specified bandwidth, duration, and a destination IP. Each data flow is produced by a SOHO user and sent to a specific external user as specified by the destination IP of the data flow. Control flows are used internally by SOHOSim to measure network quality. SOHOSim has a single router entity that manages network flows, both data and

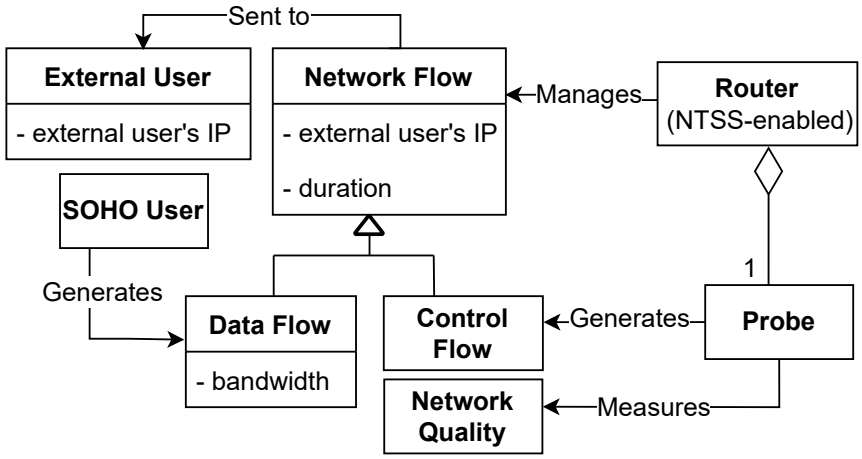


Figure 3.5: Conceptual view of our NTSS simulator, SOHOSim.

control. The router is configured to use an NTSS to separate the flows into different classes and ensure high quality network connectivity. Each class is assigned a priority level, and the NTSS ensures that the appropriate share of network bandwidth is allocated to each traffic class, maintaining high-quality connectivity across the network. The *probe* entity, which operates on the router, generates control flows and measures the quality metrics. Control flows are low-bandwidth, sent by the probe and received by the same external users that receive data flows. In contrast to data flows, control flows are echoed back to the router so that the probe can compute quality metrics such as latency, loss and jitter. For control flows, the bandwidth is fixed and set to a low value. The duration and the destination of control flows are the same as those of data flows.

To use SOHOSim for testing an NTSS, we generate parallel data flows from SOHO users to different external users such that all flows pass through the router. The NTSS installed on our (virtual) router then divides these flows into separate classes. To ensure realism in all our experiments in this thesis, we ensure that the total bandwidth of data

flows does not exceed the system's capacity. Further, we consider different traffic profiles – for example, small, frequent UDP packets for VoIP traffic, and larger, bursty TCP flows for background traffic such as file transfers.

Using SOHOSim, we assess the network quality for each NTSS class, with quality metrics including latency, jitter, and loss. In Chapter 5, we introduce an aggregated measure that combines these network quality metrics into a single value representing overall network performance. Similar to fitness functions in Simulink models and ADS, we can establish a threshold for this aggregated measure to determine whether the network quality for a given test input is acceptable. Further details on how we compute network quality are provided in Chapter 5. Further, in Chapter 5, we will evaluate the accuracy of SOHOSim with a hardware-in-the-loop testbed of NTSS.

3.4 Summary

This chapter details the case-study systems used to evaluate the approaches proposed in the thesis, covering domains such as industrial-scale Simulink models, ADS, and networking systems. Section 3.1 introduces Simulink models, which are designed for simulation and analysis of CPS, and describes the inputs and outputs of these models. In particular, fitness functions are employed to evaluate test outcomes for Simulink models. Section 3.2 provides details on the ADS controllers, including both PID-based and DNN-based end-to-end controllers, which are tested in the BeamNG simulator using different driving environments. Section 3.3 introduces our NTSS simulator that generates outbound traffic streams in a SOHO setup and measures the quality of the network. In the following chapters, we present

our approaches to address the effectiveness and reliability challenges of CPS testing and evaluate them using the case-study systems presented in this chapter.

Chapter 4

Test Generation Strategies for Building Failure Models and Explaining Spurious Failures

In this chapter, we present a framework for generating failure models to identify spurious failures in compute-intensive CPS. Spurious failures pose a significant challenge for testing. Our approach leverages ML-based test generation to efficiently and effectively explore the input space. From the generated data, we build interpretable failure models using decision-rule learning, providing human-understandable rules that characterize the conditions leading to failures. We evaluate our framework on case studies presented in Chapter 3, demonstrating its effectiveness in producing accurate failure models that domain experts can validate to identify spurious failures, ultimately improving testing effectiveness.

4.1 Introduction

Traditionally, software testing has been concerned with finding inputs that reveal failures in the system under test (SUT). However, failures do not always indicate faults in the SUT. Instead, failures may arise because the test inputs are *invalid* or *unrealistic*. For example, in an airplane autopilot system, a failure indicating that the ascent requirement fails for a test input that points the plane nose downward would be invalid. This is because the failure is caused by an unmet environment assumption: the nose should be upward for ascent. For another example, consider a network-management system. In such a system, quality-of-service requirements will inevitably fail for unrealistic test inputs that overwhelm the network beyond its capacity. Environment assumptions capture the expected conditions for a system's operational environment [GPB02]. Attempting to test a system for a more general environment than its expected operational environment may lead to overly pessimistic testing and verification results. We refer to failures arising from test inputs violating the system's environment assumptions as *spurious failures*. It is often the case that environment assumptions are not fully known for software systems [SMP+18]; therefore, it is difficult to determine whether a failure is indeed spurious.

Automated random testing (fuzzing) [MFS90] becomes more effective in exercising the main functions of a system if the fuzzer avoids spurious failures [GKH+20]. Spurious failures particularly pose a challenge for CPS simulators, where a single test execution takes significant time to complete. For compute-intensive (CI) CPS, we want to use the limited testing time budget to generate valid inputs that exercise the system's main functions. A promising approach for identifying spurious failures is to build *failure models* [KTT17,

KHSZ20, KPAB22]. Failure models provide conditions that explain the circumstances of failures and describe when a failure occurs and when it does not [KHSZ20]. Failure models can infer rules leading to and only to failures. These rules are candidates to be validated against domain knowledge to determine whether the failures that the rules identify are spurious.

Recent research on synthesizing input grammars [KTT17, KLS21, BSAL17, AFH+19, WCWL19] and abstracting failure-inducing inputs [KPAB22, GKH+20, KHSZ20] aims to understand the circumstances of different failures. These approaches start from an example failure and iteratively generate more tests to learn the input conditions that lead to that failure. The tests are generated via fuzz testing with or without an input grammar. These approaches are geared towards systems with string inputs, where oracles are typically binary (pass/fail) verdicts. However, these approaches are not optimized for systems with numeric inputs, where the inputs are not governed by grammars and where quantitative fitness functions, developed based on system requirements, are used to determine the degree to which test inputs pass or fail. These quantitative fitness functions enable exploration of input space using multiple test-generation heuristics and learning algorithms, resulting in test sets with sufficient information to infer candidate rules for identifying spurious failures.

In this chapter, we propose a framework to infer failure models for compute-intensive (CI) CPS systems. We follow a data-driven approach and infer failure models by harvesting information from a set of test inputs. To generate such sets, one can use either explorative or exploitative search methods [Luk13]. Recall from Section 1.3 that explorative search attempts to sample the entire search space, whereas exploitative search attempts to sample the most informative regions of the search space. The challenge with the explorative

approach is that we need to collect and execute many test inputs from the search space to determine if they pass or fail. For CI systems, this takes significant time and may become infeasible. The challenge with the exploitative approach is that one needs effective guidance for sampling within large and multi-dimensional search spaces.

ML has been used for improving the effectiveness and efficiency of both explorative and exploitative search [GMN+20, LSN+22, BANBS16, MNB17, MNBP20, THMY21, JS02]. For explorative search, surrogate-assisted test generation relies on ML to predict verdicts for test inputs instead of executing them all [Jin05, BANBS16, NSS+23, MNB17]. Using a quantitative surrogate, one can forego system executions when the predicted verdicts remain valid after offsetting prediction errors. Otherwise, we execute the SUT and use the results from the executed test inputs to refine the surrogate. As for exploitative search, ML-guided test generation aims to infer boundary regions that separate passing and failing test inputs and to subsequently sample test inputs from those regions [GMN+20, LSN+22]. The intuition is that tests sampled in the boundary regions are more informative for identifying failures and can be used to further refine the boundary regions. Both approaches provide a set of labelled test inputs from which one can infer failure models using techniques such as decision-rule learning [WFH11]. Human experts must nonetheless review and validate the resulting rules to determine whether they represent genuine spuriousness. The use of interpretable ML techniques such as decision-rule learning allows failure models to be expressed as easily understandable rules linked to system inputs, making them ideal for human interpretation.

While surrogate-assisted and ML-guided test generation algorithms have been previously used to generate individual test inputs [BANBS16, NSS+23] , their efficacy in gen-

erating failure models remains unexplored. Specifically, earlier work strands [BANBS16, NSS+23] employ ML to more effectively steer test generation towards areas within the search space that are likely to contain the most severe failures. In this chapter, we use ML to devise new test generation algorithms, with the aim of inferring failure models for systems that have numeric inputs. We evaluate the resulting failure models against those produced by baselines. Our evaluation answers two main questions: (1) How accurate are the failure models generated by the surrogate-assisted and ML-guided techniques in predicting failures? (2) How useful are failure models for identifying spurious failures? We use two kinds of case-study systems in our evaluation: (i) Four CPS Simulink models with 12 requirements from Table 3.1 in Section 3.1 that are non-compute intensive (non-CI). (ii) Two industrial CI systems: autopilot Simulink model in Table 3.1, and the NTSS simulator we discussed in Section 3.3. In summary, we make the following contributions:

(1) We propose a data-driven framework for inferring failure models for systems with numeric inputs including CPS and network systems (Section 4.3).

(2) We propose a dynamic surrogate-assisted algorithm that uses multiple surrogate models simultaneously during search, and dynamically selects the prediction from the most accurate model (Section 4.3.2). Our evaluation performed based on seven surrogate-model types in the literature [NSS+23, THMY21, DMTBZTL16, DAB21] shows that, compared to using surrogate models individually, our dynamic surrogate-assisted algorithm provides the best trade-off between accuracy and efficiency by generating datasets that are at least 33% larger while being at least 28% more accurate (RQ1 in Section 4.4.2).

(3) We compare the accuracy of failure models obtained using our dynamic surrogate-assisted approach against two ML-guided techniques as well as two baselines. One baseline is random-search, and the other is an adaptation of a state-of-the-art approach that generates failure models for systems with structured inputs [KHSZ20]. Our results show that our dynamic surrogate-assisted algorithm yields failure models with an average accuracy, precision, and recall of 83%, 72%, and 88%, respectively, significantly outperforming the ML-guided algorithms and the baselines (RQ2 in Section 4.4.3 and RQ3 in Section 4.4.4).

(4) We demonstrate that failure models built using our dynamic surrogate-assisted algorithm generate useful rules for identifying spurious failures in our CI systems, as validated by domain knowledge (RQ4 in Section 4.4.5).

(5) We present lessons learned based on our findings: The first lesson summarizes the advantages of using decision rules for building failure models. The second lesson highlights the limitations of focusing testing on finding individual failures and why failure models provide better insights about the effectiveness of testing algorithms.

It is essential for systems to handle all potential inputs including those that violate environment assumptions, and hence, are invalid. Spurious failures caused by invalid test inputs indicate a need for additional safeguards against invalid inputs that may be generated, among other sources, by human operator errors or malfunctioning hardware components, such as inaccurate sensor data. However, these invalid test inputs do not exercise the core functionality of a system. While ensuring that a given system is safeguarded against invalid inputs is crucial, the inability to identify spurious failures can distort our understanding of the system's capabilities. This may also lead to misplaced confidence in

a testing strategy that reveals numerous failures, yet offers little insight into the system's primary functions.

Structure. Section 4.2 motivates the need for identifying spurious failures. Section 4.3 presents our data-driven framework for inferring failure models and presents alternative surrogate-assisted and ML-guide algorithms for building failure models. Section 4.4 presents an evaluation of these algorithms. Section 4.5 outlines the main lessons learned from the research. Section 4.6 summarizes the chapter.

4.2 Motivation

Using two real-world CI systems, we motivate the need for identifying spurious failures. These systems, both of which are open-source, include an NTSS we discussed in Section 3.3 and an autopilot system from Table 3.1.

Recall from Section 3.3 that NTSS is typically deployed on routers to ensure high network performance for real-time streaming applications such as teleconferencing (e.g., Zoom). This has made systematic testing of NTSS essential as a way to ensure that networks meet their quality-of-experience requirements. NTSS works by dividing the total network bandwidth into classes with different priorities. To test the performance of an NTSS, we assign data flows with different bandwidth values to different NTSS classes. The purpose is to ensure that NTSS is configured optimally and can maintain good performance even when a high volume of traffic flows through its different classes. When we stress-test an NTSS, no matter how well-designed the NTSS is, we expect the quality of experience

to deteriorate and become unacceptable eventually. Test inputs that stress NTSS beyond a certain limit deterministically fail and do not help reveal flaws or suboptimality in the NTSS design. Our approach in this chapter infers the limit on the traffic that can flow through different NTSS classes without compromising the quality of experience. For an NTSS setup with eight classes from class0 to class7, we learn the following rule specifying failing test inputs:

r1: IF (class5 + class6 + class7 > 0.75 · threshold) THEN FAIL

In the above rule, threshold is the sum of the maximum bandwidths of classes 5, 6, and 7. As we discuss in Section 4.4.5, we validate Rule r1 with a domain expert and confirm that failures specified by this rule are indeed spurious. Rule r1 indicates that attempting to simultaneously utilize classes 5, 6 and 7 more than 75% of their maximum ranges would compromise quality of experience for the entire network. Rule r1 helps domain experts in at least two ways: (1) it informs them that test inputs that satisfy the rule are spurious, since such test cases do not reveal design faults, and (2) it provides experts with data-driven evidence that the cumulative utilization of classes 5, 6, and 7 should be kept below the identified limit.

Our second case study is an autopilot system from Lockheed Martin's benchmark of challenge Simulink models [Cha17]. This autopilot system is expected to satisfy the following requirement:

φ= "*When the autopilot is enabled, the aircraft should reach the desired altitude within 500 seconds in calm air*".

When we test the autopilot by fuzzing, we find several test inputs that violate this requirement and several test inputs that satisfy it. It is however unclear whether the failures are due to faults in the system or due to missing or unknown assumptions on the system inputs. Failures caused by missing or unknown assumptions would be spurious. As we discuss in Section 4.4.5, we identify the following rule as one that indicates spurious failures:

r2: IF (PitchWheel[0..300] $\leq$ -28 $\wedge$ Throttle[0..300] $\leq$ 0.1) THEN FAIL

Here, Throttle[0..300] is the boost applied to the engine by the pilot during the first 300s, and PitchWheel[0..300] is the upward or downward degree of the aircraft nose, again during the first 300s. Note that both Throttle and PitchWheel are signals over time. In order to validate r2, we examined the handbook of the De Havilland Beaver aircraft [(AS09]. According to the handbook, for this aircraft type, to satisfy requirement φ, the pilot should manually adjust the throttle boost (Throttle) to a sufficiently high value. The handbook further states that to be able to ascend, the plane's nose should not be pointing downward. That is, r2 describes a situation where the pre-conditions for φ are not met. Hence, the tests in these ranges are expected to fail and are uninteresting for revealing system faults. Further, r2 can be used for implementing safeguards against misuse by the human operator (pilot).

4.3 Generating Failure Models

Figure 4.1 shows our framework for generating failure models. The inputs to our framework are: (1) an executable system or simulator S, (2) the input-space representation $\mathcal{R}$ for S,

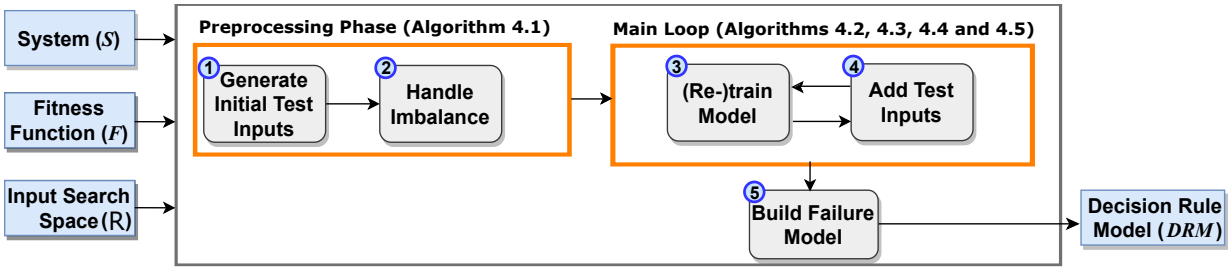


Figure 4.1: Our framework for generating failure models. The main loop of the framework, i.e., steps 3 and 4, can be realized using two alternative test-generation strategies: (1) Surrogate-assisted test generation, or (2) ML-guided test generation. For surrogate-assisted test generation, one can use either Algorithm 4.2, which is based on an individual surrogate model, or Algorithm 4.3, which is based on our proposed dynamic model. For ML-guided test generation, one can use either Algorithm 4.4, which uses regression trees to identify the boundary regions between passing and failing test cases, or Algorithm 4.5, which uses logistic regression for the same purpose.

and (3) a quantitative fitness function F for each requirement of S; an example requirement, φ, for autopilot was given in Section 4.2. The full set of requirements for our case studies is available in our supplementary material [req25]. We make the following assumptions about the search input space ($\mathcal{R}$) and the fitness function (F):

- **A1.** We assume that the system inputs are variables of type real or enumerate. For each real variable, the range of the values that the variable can take is bounded by an upper bound and a lower bound.
- **A2.** For each requirement of S, we have a fitness function F based on which a pass/fail verdict can be derived for any test input.

Assumptions **A1** and **A2** are common for CPS models expressed in Simulink [Cha17, MNGB19, TFP+19, AWM+19], automated driving systems [NSS+23], and network-management systems [JNSS23], and are valid for all the case studies we use in our evaluation.

Algorithm 4.1 The preprocessing phase of Figure 4.1

Input S: System
Input $\mathcal{R} = \{R_1, \ldots, R_n\}$: Ranges for input variables v_1 to v_n
Input F: Fitness Function
Input d: The budgeted dataset size
Output DS: A set of test inputs and their fitness values

1: $DS^i \leftarrow$ GenerateTests($\mathcal{R}$, $d/2$); //(Adaptive) Random Testing
2: $DS^l \leftarrow$ Execute(S, DS^i, F); //Compute fitness values
3: $DS^b \leftarrow$ HandleImbalance(DS^l); //Use SMOTE to generate synthetic test inputs
4: $DS^{b,l} \leftarrow$ Execute(S, DS^b, F); //Compute fitness values of the tests generated by SMOTE
5: $DS \leftarrow DS^{b,l} \cup DS^l$; //Combine the test inputs generated by adaptive random testing and SMOTE along with their fitness values to form a dataset
6: **return** DS

As discussed in Section 4.1, we examine two alternative test-generation approaches for building failure models: surrogate-assisted and ML-guided. Both approaches can be captured using the framework shown in Figure 4.1: The preprocessing phase generates a set of test inputs labelled with fitness values. The main loop takes the test-input set created by the preprocessing phase, and trains a model. When the framework is instantiated for surrogate-assisted test generation, the model predicts fitness values for the generated test inputs. When the framework is instantiated for ML-guided test generation, the model guides test-input sampling. The main loop extends the test-input set using the trained model while also refining the model based on newly generated tests. After the main loop terminates, the framework uses the test-input set to train, using decision-rule learners, a failure model. In the remainder of this section, we detail each step of the framework shown in Figure 4.1.

4.3.1 Preprocessing Phase

Algorithm 4.1 describes the preprocessing phase that generates the initial dataset for training a model to be used in the main loop of Figure 4.1. This algorithm first randomly generates half ($d/2$) of the budgeted test inputs and computes a fitness value for each test input by executing the test input using S (lines 1-2). Since ML models perform poorly when the training set is imbalanced, we attempt to address any potential imbalance before using the data for ML training [WFH11] (line 3). In our work, the imbalance, if one exists, is between the pass and the fail classes. We use the well-known synthetic minority over-sampling technique (SMOTE) for addressing imbalance [CBHK02]. Let *minor* (resp. *major*) be the number of tests in the minority (resp. majority) class. SMOTE over-samples the minority class by taking each minority-class sample and introducing synthetic examples along the line segments joining any/all of the k minority-class nearest neighbours [CBHK02]. The process is repeated until we have $m = major - minor$ new such tests.

We discard the labels from SMOTE and instead execute the tests to compute their actual fitness values (line 4). We discard the labels provided by SMOTE, since SMOTE categorizes test inputs as pass or fail. Instead, we require test inputs to be labelled with their quantitative fitness values; this enables us to train regression ML models in Step 3 of our approach in Figure 4.1. The final dataset (DS) is returned at the end (line 6). Although not shown in Algorithm 4.1, to have exactly d tests in DS, we generate the remaining $(d/2 - m)$ tests randomly. Generating these remaining tests randomly does not introduce a new imbalance problem because if random test generation leads to major imbalance, then m is already close to $d/2$ and only a few additional tests need to be

generated. Otherwise, a small m indicates that random testing is relatively balanced; in that case, no special provision is necessary for imbalance mitigation in the randomly generated datatset. Since we discard the labels generated by SMOTE and compute the actual labels, the imbalance problem may in principle persist even after applying SMOTE. For our experiments (Section 4.4), most synthetic samples generated by SMOTE indeed belong to the minority class. Our preprocessing therefore successfully addresses imbalance in our case studies.

4.3.2 Main Loop

The main data-generation loop is realized via two alternative algorithms, described below: surrogate-assisted and ML-guided.

The goal of this approach is to use surrogates to predict fitness values for some test inputs and thus not execute the system for all test inputs. Hence, surrogates help explore a larger portion of the input space and generate larger test sets.

4.3.2.1 Surrogate-Assisted Test Generation

Figure 4.2 illustrates the surrogate-assisted test generation process, which takes the same inputs as the framework in Figure 4.1. The output is a labelled dataset, DS, used in Step 5 of Figure 4.1 to build failure models. The procedure in Figure 4.2 is as follows:

1. Start with preprocessing (Algorithm 4.1) to produce an initial dataset (Step 1 of Figure 4.2).

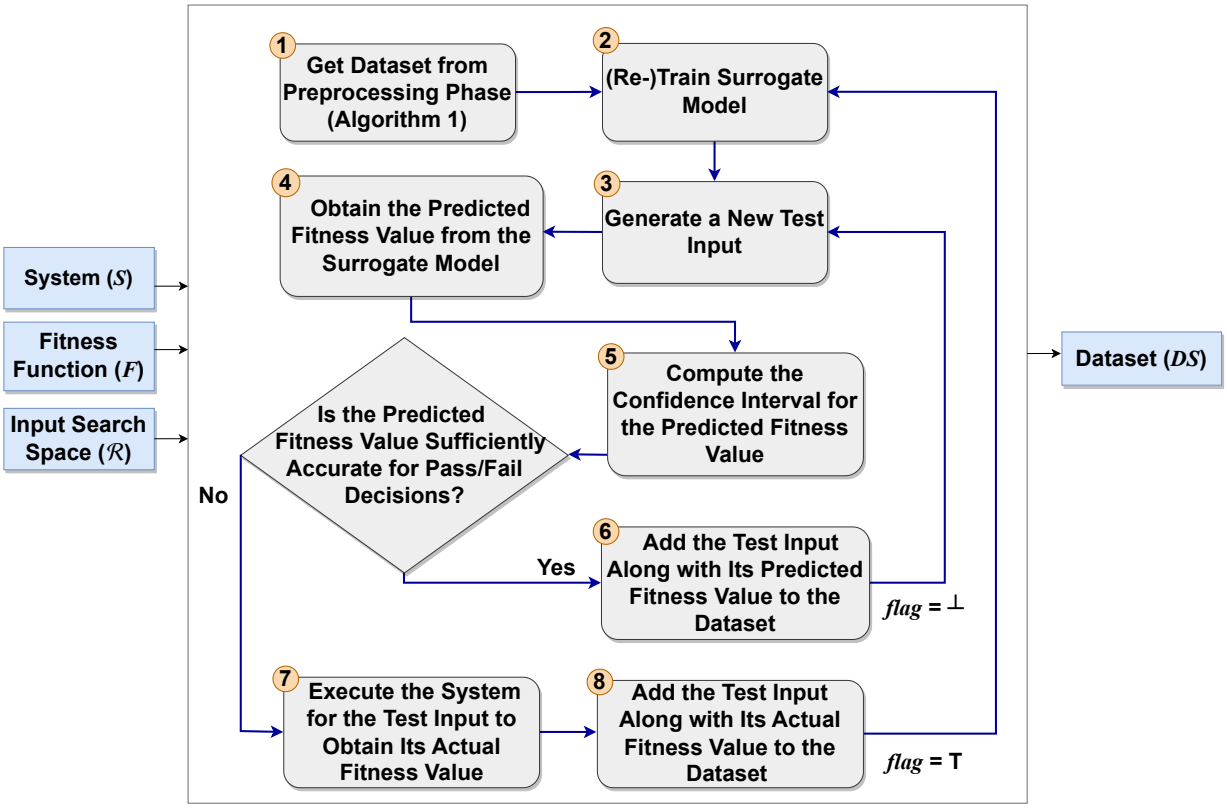


Figure 4.2: Illustration of the workflow of Algorithm 4.2

2. Train a surrogate model with the initial dataset (Step 2 of Figure 4.2).

3. Generate a new test input (Step 3 of Figure 4.2) and predict its fitness value using the surrogate model (Step 4 of Figure 4.2).

4. Calculate a confidence interval for the predicted fitness value (Step 5 of Figure 4.2).

5. If the prediction is not accurate, run system S to obtain the actual fitness value (Step 7 of Figure 4.2), add the test input along with its actual fitness value to the

dataset (Step 8 of Figure 4.2), and return to Step 2 of Figure 4.2 to re-train the surrogate model.

6. If the prediction is accurate; add the test input along with its predicted fitness value to the dataset (Step 6 of Figure 4.2); and, return to Step 3 of Figure 4.2.

Note that if we execute system S for a test input, as mentioned in item (5) above, the surrogate model is retrained using the dataset updated with this new test execution. In contrast, if system execution is not required, as in item (6) above, retraining the surrogate model is unnecessary.

To demonstrate the calculation of the confidence interval described in item (4) above, consider the example provided in Figure 4.3. In this figure, two sample test inputs, denoted as t and t', are shown. Suppose the predicted fitness values are $\bar{F}(t) = 8$ and $\bar{F}(t') = -1$, indicating a pass label for t and a fail label for t', respectively. Assume that the prediction error, e, of the surrogate model is 2. The confidence interval is calculated as $\bar{F} \pm e$. That is, the confidence interval for $\bar{F}(t)$ is $[6, 10]$, while the confidence interval for $\bar{F}(t')$ is $[-3, 1]$. Using the confidence intervals, we determine whether to execute system S for t and t'. For input t, the confidence interval of $\bar{F}(t)$ falls entirely within the positive range. Hence, even after accounting for error, we still label the test input t as a pass. Therefore, there is no need for system S to be executed for input t, since t can be confidently labelled as pass (item (6) above). However, for input t' the confidence interval of $\bar{F}(t')$ spans both positive and negative values. This indicates that a label cannot be confidently assigned to t'. As a result, system S needs to be executed for t' to obtain its actual fitness value and an accurate verdict (item (5)).

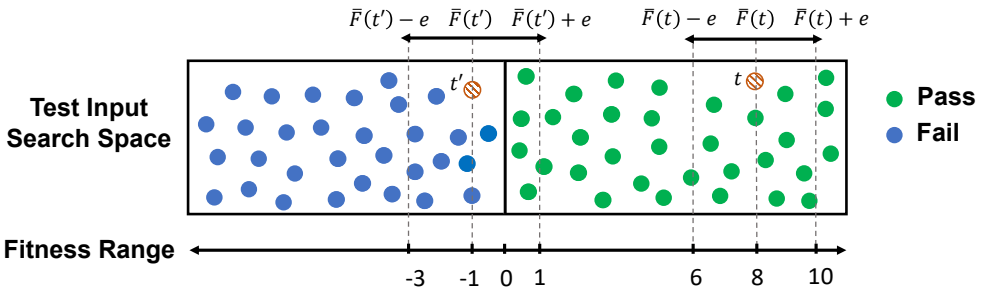


Figure 4.3: Illustration of how to calculate the confidence interval for predicted fitness values to determine whether a predicted fitness value is accurate. Specifically, the figure shows confidence intervals, i.e., $\bar{F} \pm e$, of the predicted fitness values for two test inputs t and t'. For the test input t where the predicted fitness confidence interval remains entirely in the positive range (i.e., the verdict of the test inputs in this range is pass), we do not execute the system since we can say that the test input is passing even after accounting for error e. However, for the test input t', the predicted fitness confidence interval spans both positive and negative values (i.e., it includes test inputs with both pass and fail verdicts). Therefore, we need to execute the system for t' to obtain its actual fitness value.

The pseudo code of the surrogate-assisted test generation is provided in Algorithm 4.2. Below, we discuss each line of this algorithm in detail.

Line 1 of Algorithm 4.2 uses Algorithm 4.1 to obtain an initial dataset, DS:

1: $DS \leftarrow$ Preprocessing(S, $\mathcal{R}$, F, InitialDatasetSize);

On line 2, DS^l is created to maintain a record of the test inputs for which S is executed. We use this dataset to train a surrogate model SM and compute its error e (line 5):

2: $DS^l \leftarrow DS$; $flag \leftarrow \top$;

5: $(SM, e) \leftarrow$ Train(DS^l);

Specifically, we train SM using 80% of DS^l and compute the mean absolute error of SM on the remaining 20% of DS^l. The split ratio is based on the well-known 80/20 rule [PKKS19]. We employ a boolean variable, $flag$, to decide whether training a surrogate

Algorithm 4.2 Test generation using surrogates

Input S: System
Input $\mathcal{R} = \{R_1, \ldots, R_n\}$: Ranges for input variables v_1 to v_n
Input F: Fitness Function
Input InitialDatasetSize: size of initial dataset
Output DS: A dataset to train failure models

1: $DS \leftarrow$ Preprocessing(S, $\mathcal{R}$, F, InitialDatasetSize); //Run Algorithm 1 to generate an initial dataset
2: $DS^l \leftarrow DS$; $flag \leftarrow \top$;
3: **while** (execution budget remains) **do**
4: **if** ($flag$)
5: (SM, e) $\leftarrow$ Train(DS^l); //Training SM; e is the error
6: **end**
7: $t \leftarrow$ GenerateTests($\mathcal{R}$, 1); //Generate one test input
8: $\bar{F}(t) \leftarrow SM(t)$; // Use SM to predict a fitness value for t
9: **if** ($\exists \sim \in \{\geq, <\} \cdot (\bar{F}(t) \sim 0) \wedge (\bar{F}(t) \pm e \sim 0)$ //Calculate the confidence interval of the predicted fitness value to decide whether system S needs to be executed for t or not
10: $DS \leftarrow DS \cup \{\langle t, \bar{F}(t)\rangle\}$; //Add the test input along with its predicted fitness value to the dataset
11: $flag \leftarrow \bot$; //No SM re-training is needed
12: **else**
13: $\{\langle t, F(t)\rangle\} \leftarrow$ Execute(S,{t}, F); //Obtain the actual fitness value of t by executing system S
14: $DS^l \leftarrow DS^l \cup \{\langle t, F(t)\rangle\}$; //Add the test input along with its actual fitness value to the dataset
15: $DS \leftarrow DS \cup \{\langle t, F(t)\rangle\}$; $flag \leftarrow \top$; //SM re-training is needed
16: **end**
17: **end**
18: **return** DS

model is necessary. Initially, on line 2, we initialize *flag* to $\top$ to ensure that a surrogate model is trained during the first iteration. Then, on line 7, we randomly generate a new test input and denote it by t:

7: $t \leftarrow$ GenerateTests($\mathcal{R}$, 1);

Then, on line 8, the surrogate model SM predicts a fitness value for t. The predicted fitness value is denoted by $\bar{F}(t)$:

8: $\bar{F}(t) \leftarrow SM(t)$;

On line 9, we calculate a confidence interval for $\bar{F}(t)$ based on the prediction error e. Specifically, we compute the interval $[\bar{F}(t)-e, \bar{F}(t)+e]$ as the confidence interval for $\bar{F}(t)$. If the confidence interval remains entirely within the positive or negative range, indicating a definite pass or fail verdict for the test input t, we skip executing system S for the test input t. This is because we can confidently label it as either pass or fail. Subsequently, we add the test input t along with its predicted fitness value (i.e., $\bar{F}(t)$) to DS (line 10). In addition, we set *flag* to $\perp$ in order to prevent retraining the surrogate model SM in the next iteration (lines 11):

9: **if** $\exists \sim \in \{\geq, <\} \cdot (\bar{F}(t) \sim 0) \wedge (\bar{F}(t) \pm e \sim 0)$

10: $DS \leftarrow DS \cup \{\langle t, \bar{F}(t)\rangle\}$;

11: *flag* $\leftarrow \perp$;

Otherwise, on line 12, if the confidence interval spans both positive and negative numbers, indicating that it covers test inputs with both pass and fail verdicts, we proceed to execute system S to calculate the actual fitness value for t. Then, we add t along with its actual fitness value to DS and DS^l (lines 14-15). In this case, we set *flag* to $\top$ to indicate that a system execution has taken place (line 15):

12: **else**

13: $\{\langle t, F(t)\rangle\} \leftarrow$ Execute(S,$\{t\}$, F);

14: $DS^l \leftarrow DS^l \cup \{\langle t, F(t)\rangle\}$;

15: $DS \leftarrow DS \cup \{\langle t, F(t)\rangle\}$; *flag* $\leftarrow \top$;

In the latter case (i.e., lines 12-15), we re-train the surrogate model SM using the dataset DS^l which includes the new test input and its actual fitness value (lines 4-5). The algorithm

returns the final dataset DS when the execution budget runs out (line 18):

18: **return** DS;

The execution budget expires when either the system has been executed to the point where the size of DS^I reaches its desired limit, or our time budget is exhausted, depending on which occurs first.

In our experimentation (Section 4.4), we consider the surrogate-model types shown in Table 4.1. These surrogate-model types are the most widely used ones in the evolutionary search and software testing literature [NSS+23, THMY21, DMTBZTL16, DAB21]. We select these models because they represent a diverse set of learning paradigms, ranging from probabilistic models and ensemble-based learners to neural models and kernel-based approaches. This diversity ensures coverage of models that are known to balance predictive accuracy, computational efficiency, and generalization ability, which are essential characteristics for effective surrogate modelling. As suggested by the literature and also as we show in our evaluation (Section 4.4.2), no surrogate-model type consistently outperforms the others [XZLX21, FBBE16]. Therefore, it is recommended to use a combination of surrogate models [HLF22]. In this chapter, we propose, to our knowledge, a novel variation of Algorithm 4.2 where we train multiple surrogate models and use for predicting fitness values the model that has the lowest error. This variation is shown in Algorithm 4.3 where we change line 5 of Algorithm 4.2 to train and tune a list of surrogate models instead of just one model. We then select the surrogate model with the lowest error for making predictions until the next time we re-train the models. Similarly, each time we execute line 5 of Algorithm 4.2, we re-train a list of surrogate models and select the one that has

Table 4.1: Surrogate models and their descriptions.

Name	Description	Name	Description
GL	Gaussian Process Regression – nonparametric Bayesian with linear kernel.	RT	regression tree.
GNL	Gaussian Process Regression – nonparametric Bayesian with nonlinear kernel.	RF	random forest.
LSB	Gradient Boosting – an ensemble of regression trees.	SVR	Support Vector Regression.
NN	a two-layer feedforward Neural Network.		

Algorithm 4.3 Dynamically selecting surrogates

4: ...
5: **for** i = 1 to sm **do** //Train surrogate models $SM_1, ..., SM_{sm}$
6: $(SM_i, e_i) \leftarrow \text{Train}(DS^l)$;
7: **end**
8: $(SM, e) \leftarrow$ Select $SM \in \{SM_1, \ldots, SM_{sm}\}$ with the lowest error e
9: ...

the lowest error. We refer to our proposed variation as *dynamic* surrogate-assisted test generation.

In both Algorithm 4.2 and the variation suggested in Algorithm 4.3, the first time we train a surrogate model, we also tune its hyperparameters using Bayesian optimization [SLA12]. We use the same tuned hyperparameters in all future iterations. The cost of training and tuning surrogate models for the first time is on the same scale as the cost of a single execution of our CI systems. The time for subsequent re-training of surrogate models is nonetheless negligible since re-training does not involve any tuning. As we discuss in Section 4.4, the overhead of re-training surrogate models does not deteriorate performance compared to other alternatives.

4.3.2.2 ML-Guided Test Generation

ML-guided test generation uses ML models for identifying the boundary regions that discriminate pass and fail test inputs and iteratively concentrating test-input sampling to those regions. The idea is that, irrespective of the separability of the set of test inputs, ML models can shift the focus of sampling from the homogeneous regions where either fail or pass verdicts are scarce to regions where neither fail nor pass would be dominant. We consider two alternative ML models that can help us sample from such boundary regions: regression trees (Algorithm 4.4) and logistic regression (Algorithm 4.5). Regression tree excels at capturing intricate, non-linear relationships without prior assumptions about the boundary's shape. Logistic regression provides a computationally efficient and stable method to converge on a smooth decision boundary, which is particularly valuable for guiding the sampling process when the boundary is roughly linear or the feature space is large. As we describe below, a regression tree approximates pass-fail boundaries in terms of predicates over inputs variables, while logistic regression infers a linear formula over input variables.

Regression-Tree Guided Test Generation. Algorithm 4.4 uses the *DS* dataset obtained from Algorithm 4.1 (the preprocessing phase) to train a regression-tree model (lines 1-3). In our regression-tree models, tree edges are labelled with predicates $v_i \sim c$ such that v_i is an input variable, $c \in \mathbb{R}$ is a constant and $\sim \in \{\leq, >\}$. The tree leaves partition the given dataset into subsets such that information gain is maximized [WFH11]. Each leaf is labelled with the average of the fitness measures of the test inputs in that leaf. Provided with a regression tree, Algorithm 4.4 identifies predicates $\{v_{i_1} \sim c_{i_1}, \ldots, v_{i_m} \sim c_{i_m}\}$

that appear on the two paths whose leaf-node values are closest to zero (one above and one below zero). These predicates specify the boundary between pass and fail, and, as such, we call them *boundary* predicates. By simplifying the boundary predicates, each variable can have at most one upper-bound predicate ($v \leq c$) and at most one lower-bound predicate ($v > c$). For each predicate $v_{i_j} \sim c$ where $\sim \in \{\leq, >\}$, the algorithm replaces the existing range R_{i_j} of v_{i_j} with $R'_{i_j} = [c{-}5\% \cdot c, c{+}5\% \cdot c]$ (lines 5-7). This will ensure that we sample v_{i_j} within the 5% margin around the constant c. The variables that do not appear in the boundary predicates retain their range from the previous iteration. We note that if R'_{i_j} does not reduce the range for v_{i_j}, i.e., the size of R'_{i_j} is greater than R_{i_j}, we do not replace R_{i_j} with R'_{i_j}. This is to ensure that larger ranges are not carried over to the next iteration if the range has already been narrowed at some previous iteration. Next, the algorithm generates a test input within the constrained search space (line 8), executes the test input, and adds it along with its fitness measure to *DS* (line 9). The algorithm returns the final dataset after the execution budget runs out (line 11). The execution budget expires when either the system has been executed to the point where the size of DS reaches its desired limit, or our time budget is exhausted, depending on which occurs first.

To illustrate range reduction for variables using regression trees, consider the example in Figure 4.4. We choose the two thicker paths highlighted in blue since their leaf-node values are closest to zero. Variables v_1, v_2 and v_3 appear on these paths. Suppose the initial ranges for v_1, v_2 and v_3 to be $[0, 20]$, $[10, 30]$, and $[1, 7]$, respectively. Using the regression tree and the process described above, the new reduced ranges for v_1, v_2 and v_3 are $[9.5, 10.5]$, $[19, 21]$ and $[4.75, 5.25]$ respectively. By sampling within these ranges, we get to focus test-input generation on the pass-fail border identified by the regression tree.

Algorithm 4.4 ML-guided test generation with regression trees

Input S: System
Input $\mathcal{R} = \{R_1, \ldots, R_n\}$: Ranges for input variables v_1 to v_n
Input F: Fitness Function
Input InitialDatasetSize: size of initial dataset
Output DS: A dataset to train failure models

1: $DS \leftarrow$ Preprocessing(S, $\mathcal{R}$, F, InitialDatasetSize);//Run Algorithm 1 to generate an initial dataset
2: **while** (execution budget remains) **do**
3: $RegTree \leftarrow$ Train(DS);
4: Let R'_{i_1},..., R'_{i_m} be reduced ranges obtained from $RegTree$;
5: **for** each variable v_{i_j} s.t. $j \in \{1, \ldots, m\}$ **do**
6: $\mathcal{R} \leftarrow (\mathcal{R} \setminus \{R_{i_j}\}) \cup \{R'_{i_j}\}$; // Replace the range R_{i_j} of v_{i_j} in $\mathcal{R}$ with the new reduced range R'_{i_j} from line 4
7: **end**
8: $\{t\} \leftarrow$ GenerateTests($\mathcal{R}$, 1); //Generate one test input
9: $DS \leftarrow DS \cup$ Execute(S, $\{t\}$, F); // Compute fitness for t and add to DS
10: **end**
11: **return** DS

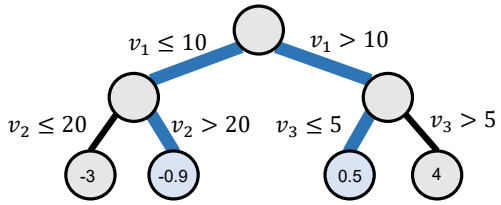


Figure 4.4: A regression tree trained in an iteration of Algorithm 4.4. The figure illustrates two regression tree paths chosen for test generation on line 4 of Algorithm 4.4.

Logistic Regression Guided Test Generation. Similar to Algorithm 4.4, Algorithm 4.5 uses the dataset DS from the preprocessing phase to train a logistic regression model (lines 1-3). Since logistic regression is a classification technique, the quantitative labels in DS are replaced with pass/fail labels before training. Recall from Section 2.1.1 that a logistic regression model is represented as $\log\left(\frac{p}{1-p}\right) = c_0 + \sum_{i=1}^{n} c_i v_i$ where $v_1, \ldots, v_n$ are the input variables, c_i's are co-efficients, and p is the probability of the pass

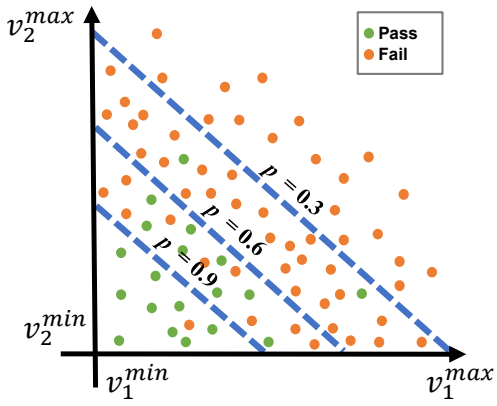


Figure 4.5: Illustrating logistic regression lines for different values of p assuming that we have two variables v_1 and v_2. The value of p is used on line 5 of Algorithm 4.5 to generate a test close to the regression border.

class [log25, WFH11]. The algorithm then randomly samples a few test inputs in the search space and picks the one closest to the logistic regression formula obtained by setting p to the percentage of the pass labels in DS (line 4-5).

By assigning a value to p in the logistic formula above, the formula turns into a linear equation. Figure 4.5 shows examples of such linear equations for different values of p assuming that we have two variables v_1 and v_2. The line with $p = 0.9$ identifies a region where the majority of test inputs pass. On the other end of the spectrum, the line with $p = 0.3$ identifies a region where the majority of test inputs fail. Setting p to the percentage of pass in DS is a heuristic to identify a region that includes a mix of pass and fail test inputs. Our sampling should, therefore, exploit this region. The test input selected on line 5 along with its fitness measure computed using S is added to DS (line 6). The algorithm re-trains the logistic regression model whenever a test input has been added to DS (line 3). The algorithm returns the final dataset when the execution budget runs

Algorithm 4.5 ML-guided test generation with logistic regression

Input S: System
Input $\mathcal{R} = \{R_1, \ldots, R_n\}$: Ranges for input variables v_1 to v_n
Input F: Fitness Function
Input InitialDatasetSize: size of initial dataset
Output DS: A dataset to train failure models

1: $DS \leftarrow$ Preprocessing(S, $\mathcal{R}$, F, InitialDatasetSize);//Run Algorithm 1 to generate an initial dataset
2: **while** (execution budget remains) **do**
3: $LogReg \leftarrow$ Train(DS);
4: $p \leftarrow$ Probability(DS); // probability of pass in DS
5: $t \leftarrow$ GenerateCloseToRegBorder($\mathcal{R}$, $LogReg$, p); // Select a test close to the regression border for p
6: $DS \leftarrow DS \cup$ Execute(S, $\{t\}$, F);// Compute fitness for t and add to DS
7: **end**
8: **return** DS

out (line 8). The execution budget expires when either the system has been executed to the point where the size of DS reaches its desired limit, or our time budget is exhausted, depending on which occurs first.

Similar to Algorithm 4.2, the hyperparameters of the regression-tree and logistic-regression models are tuned using Bayesian optimization the first time the models are built, and the same tuned hyperparameters are used in all future iterations.

4.3.3 Building Failure Models

The output of the main loop in Figure 4.1 is a set DS of tuples $\langle t, F(t) \rangle$ where t is a test input and $F(t)$ is its fitness value. We first convert DS into a dataset where test inputs are labelled by *pass* and *fail* labels. Provided with a labelled dataset, we use decision-rule models built using RIPPER [Coh95] to train failure models. Decision-rule models generate a set of *IF-condition-THEN-prediction* rules where the condition is a

conjunction of predicates over the input features and the prediction is either pass or fail. In Section 4.2, we already showed two examples of such rules for spurious failures. When no domain knowledge is available, one can directly use the input variables of the system (S) as features for learning. When domain knowledge is available, feature design for decision-rule models can be improved in two ways: (1) Excluding input variables that are orthogonal to the requirement under analysis. For example, the prerequisite for the requirement φ in Section 4.2 is that the autopilot should be enabled, i.e., APEng = on. As far as test generation for φ is concerned, we need to set APEng = on, since otherwise, φ holds vacuously. We thus do not use APEng as an input feature. For another example, in φ, we do not use the desired altitude as an input feature either, since the system is expected to satisfy φ for any desired altitude in the default range. (2) Using domain knowledge to formulate features over multiple input variables. For NTSS, as discussed in Section 4.2, the goal is to identify limits on the traffic that can flow through NTSS classes without compromising network quality. Based on domain knowledge, we know that flows have a cumulative nature. Hence, for NTSS, we use as features *sums of subsets* of flow variables. Naturally, like in any feature engineering problem, one can hypothesize alternative ways of formulating the features and empirically determine the formulation leading to highest accuracy [Ng18].

4.4 Evaluation

In this section, we evaluate our approach by answering the following research questions (RQs):

RQ1 (Configuration). *Which surrogate-assisted technique offers the best trade-off between accuracy and efficiency?* We compare *eight* surrogate-assisted algorithms. These eight algorithms are: (a) Algorithm 4.2 used with the seven surrogate models in Table 4.1 individually, and (b) the dynamic surrogate algorithm (Algorithm 4.3) that uses the seven surrogate models simultaneously and selects the best model dynamically. To measure accuracy, we check the correctness of the labels of the tests in the generated datasets; and, to measure efficiency, we evaluate the size of the generated datasets. We use the optimal algorithm for answering RQ2 to RQ4.

RQ2 (Effectiveness). *How accurate are the failure models generated by the surrogate-assisted and ML-guided techniques?* We evaluate and compare the accuracy of the failure models obtained by surrogate-assisted and ML-guided algorithms as well as those obtained based on randomly generated test inputs (random baseline).

RQ3 (SoTA Comparison). *How accurate are the failure models generated in RQ2 compared to those generated by the state of the art (SoTA)?* We use the top-performing technique from RQ2 to compare against SoTA. Among the existing approaches that build failure-inducing models [KHSZ20, GKH+20, KPAB22], we select the Alhazen framework [KHSZ20], since it uses interpretable machine learning. While Alhazen is geared towards systems with structured inputs (as opposed to systems with numeric inputs, i.e., the focus of this thesis), in the absence of baselines for systems with numeric inputs, Alhazen is our best baseline for comparison. To be able to compare with Alhazen, we adapt it to numeric-input systems, as we describe in Section 4.4.4.

Table 4.2: Names, the number of requirements and the identifiers of our study subjects. For each subject, we indicate if it is computer-intensive (CI). All artifacts including requirements statements are available in our supplementary material [req25].

Name	#Reqs	ID	CI
Tustin	9	TU1...TU9	✗
Regulator	1	REG	✗
Nonlinear Guidance	1	NLG	✗
Finite State Machine	1	FSM	✗
Autopilot	3	AP1, AP2, AP3	✓
Network Traffic Shaping System	1	NTSS	✓

RQ4 (Usefulness). *How useful are failure models for identifying spurious failures?* We answer this question for the most accurate failure models from RQ2 and for the two CI systems, namely NTSS and autopilot, discussed in Section 4.2. NTSS and autopilot are representative examples of industrial systems in the network and CPS domains, respectively. For both systems, we validate the failure-inducing rules against domain knowledge to determine whether the resulting failures are genuinely spurious.

4.4.1 Case-Study Systems

Our case-study systems are listed in Table 4.2 which are discussed in detail in Chapter 3. Below, we discuss how these systems satisfy assumptions **A1** and **A2** provided at the beginning of Section 4.3.

NTSS. To test NTSS, we transmit flows with different bandwidth values into different NTSS classes. A test input for NTSS is defined as a tuple $t = (v_1, \ldots, v_n)$ where n is the number of NTSS classes, and each variable v_i represents the bandwidth of the data flow going through class i. The fitness function for NTSS measures the network quality

based on the well-known mean opinion score (MOS) metric [SWH16]. This fitness function ensures assumption **A2** [JNSS23]. For our experiments, we use an NTSS setup running the diffserv8 mode of Common Applications Kept Enhanced (CAKE) [HJTM18, CAK25], an advanced traffic-shaping algorithm that provides 8 priority classes.

Simulink systems. As mentioned in Section 3.1, the fitness function for each requirement is encoded using RFOL semantic functions – a quantitative measure that conforms to assumption A2, as shown by Menghi et al. [MNGB19]. We note that six models in Table 3.1 were not useful for our evaluation. These six models had requirements that either did not fail, or failed for all test inputs and thus, their failure models could be trivially defined as the entire input space. Hence, in our experiments, we focus on five of the Simulink models. The first five rows of Table 4.2 list these models that have a total of 15 requirements combined.

In total, we have 16 requirements: one for NTSS, and 15 for the five Simulink models listed in Table 4.2. Among these, four are compute-intensive (CI) and twelve are non-CI. Both NTSS and autopilot are CI: On average, each execution of NTSS takes $\approx$ 4.5min, and each execution of autopilot takes $\approx$ 0.5min. The execution times for non-CI systems are negligible ($< 1s$). All experiments were conducted on a machine with a 2.5 GHz Intel Core i9 CPU and 64 GB of DDR4 memory.

4.4.2 RQ1-Configuration

We compare eight versions of the surrogate-assisted algorithm. Seven are Algorithm 4.2 used with an individual surrogate model from those in Table 4.1. We refer to each of these

algorithms as SA-XX where XX is the name of the surrogate model from Table 4.1. For example, SA-NN refers to Algorithm 4.2 used with NN. The final (i.e., eighth) algorithm is the dynamic surrogate-assisted one (Algorithm 4.3). We refer to Algorithm 4.3 as SA-DYN.

For RQ1, we use the 12 non-CI systems in Table 4.2. Performing RQ1 experiments on CI systems would be prohibitively expensive. For example, an approximate estimate for the execution time of the experiments required to answer RQ1 is over a year, if the experiments are performed on NTSS using the same experimentation platform. Thus, we opt for the non-CI systems for RQ1.

Setting. For each system, we run the eight SA-XX algorithms for an equal time budget. The time budget given for each system depends on the system execution time. The detailed time budgets are available in our supplementary material [tab25c]. To account for randomness, we repeat each algorithm for each system ten times.

Metrics. Recall that the output of the main loop in Figure 4.1 for surrogate-assisted algorithms is a set DS where the test inputs are labelled with either predicted or actual fitness values. To measure efficiency, we take the cardinality of the generated dataset, DS. Since the algorithms have the same time budget, an algorithm is more efficient than another if it generates a larger dataset. To construct failure models, we classify test inputs as pass or fail based on their fitness values. A dataset is accurate if it contains few test inputs with *incorrect labels*, i.e., test inputs with inconsistent pass/fail labels based on their predicted versus actual fitness values. To obtain the actual fitness values for all test inputs, we run the system using those test inputs for which only surrogate-assisted algorithms provided predicted fitness values. Note that this step is intended exclusively for our experiments

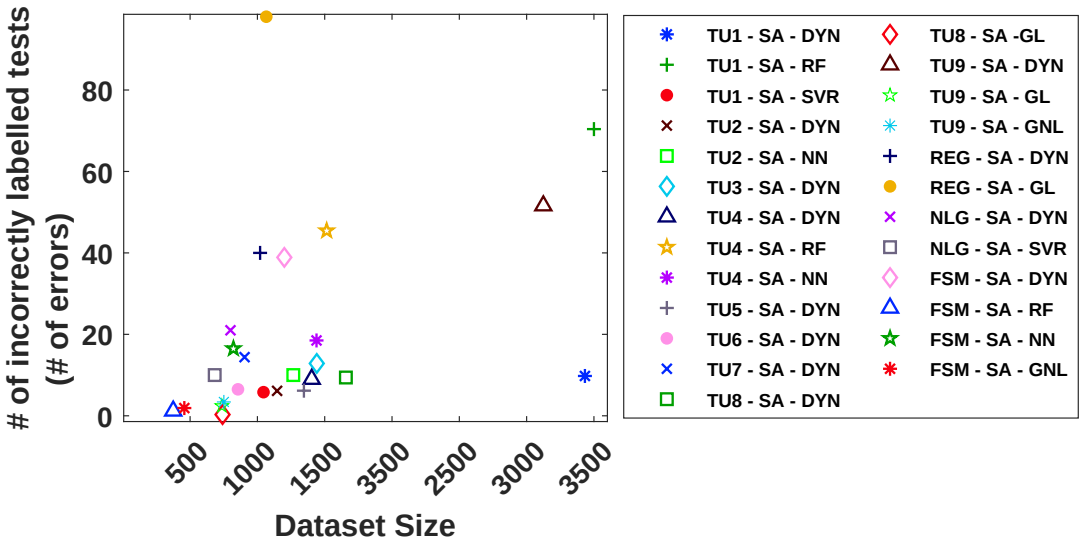


Figure 4.6: Comparing datasets generated by eight different surrogate-assisted algorithms with respect to the number of errors in the datasets and the dataset size.

and is not a component of our main approach. Next, we count the number of tests in which the predicted label differs from the actual one. The number of incorrectly labelled tests serves as a measure of error or inaccuracy for the surrogate-assisted algorithms.

Results. The scatter plot in Figure 4.6 shows the results of the experiments for RQ1. The x-axis indicates $|DS|$ and the y-axis indicates the number of incorrectly labelled tests in DS. Each point shows the result of applying one SA-XX algorithm to one system. The 12 systems are indicated by TU1 to TU9, REG, NLG, and FSM (see Table 4.2). For each system, an algorithm is considered better when it generates larger datasets with fewer errors. Since we compare eight algorithms for 12 systems, we would need 96 points to show all the results. To reduce clutter, for each system, we only show the Pareto Front (PF) points. That is, for each system, we only show the algorithms that are not dominated by other algorithms

either in terms of the number of errors or in terms of the dataset size. For example, for system TU1, algorithms SA-DYN, SA-RF and SA-SVR dominate other algorithms and offer the best trade-off between the number of errors and the dataset size. As Figure 4.6 shows, for four systems, SA-DYN is the only best trade-off (i.e., PF point), and for eight systems, it is one of the best PF points. For the latter eight cases, SA-DYN offers an alternative that, compared to the other PF points, either has considerably fewer errors while its dataset is not much smaller, or its dataset is considerably larger while the number of errors is not much higher. For example, for TU1, SA-DYN, compared to SA-RF, provides 60 less incorrectly labelled tests, while the dataset sizes are almost the same (3433 for SA-DYN vs. 3500 for SA-RF). Also, compared to SA-SVR, SA-DYN provides a larger dataset (3433 vs. 1045) while the number of incorrectly labelled tests is almost the same (10 vs. 6).

Figure 4.7 shows the ratios of the number of errors (i.e., the number of incorrectly labelled tests) over $|DS|$ for different SA algorithms and for all the 12 systems. The SA-DYN algorithm has the lowest average error which is 28% lower than that of the second best algorithm, SA-SVR, i.e., $\frac{1.99-1.43}{1.99} = 28\%$. We compare the results in Figure 4.7 using the non-parametric pairwise Wilcoxon rank-sum test [MN10] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [VD00]. The SA-DYN algorithm is statistically significantly better than other algorithms with a high effect size for GL, GNL and LSB, a medium effect size for RT, a small effect size for NN and RF, and a negligible effect size for SVR. The comparison of the dataset sizes shows that SA-DYN generates datasets that are significantly larger than those generated by SVR with a large effect size. Further, SA-DYN generates significantly larger datasets that are at least 33% larger than those obtained from other

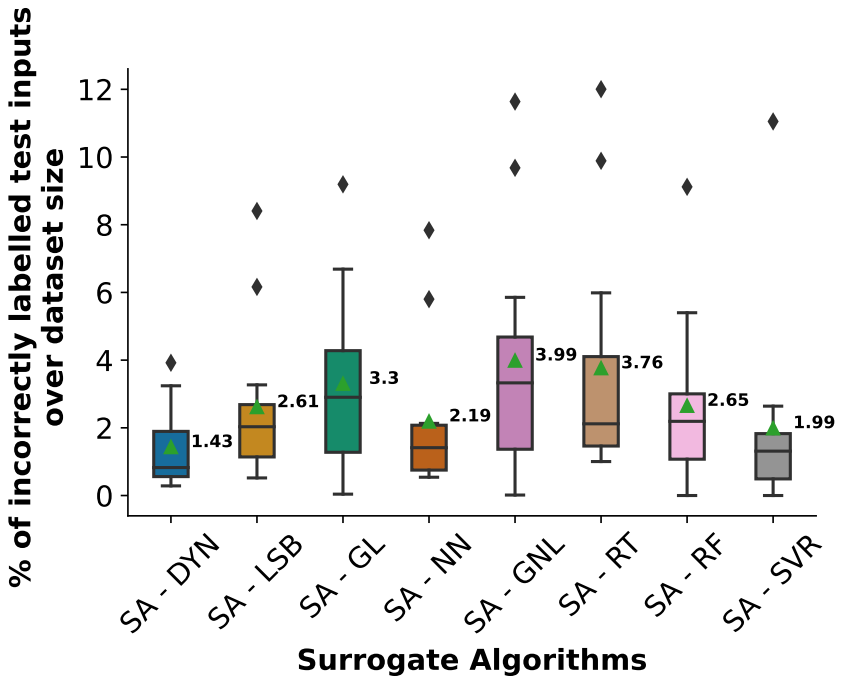


Figure 4.7: Percentages of the incorrectly labelled tests over the dataset size for different surrogate-assisted algorithms.

algorithms. Figures comparing dataset sizes and statistical tests for RQ1 are available in our supplementary material [fig25b, tab25d].

Finding. Our evaluation for RQ1 performed using seven surrogate-model types in the literature (see Table 4.1) shows that, compared to using surrogate models individually, our dynamic surrogate-assisted approach provides the best trade-off between accuracy and efficiency.

4.4.3 RQ2-Effectiveness

We compare SA-DYN, i.e., the best approach identified in RQ1, with the two ML-guided algorithms described in Section 4.3.2.2 as well as a standard adaptive random-search algorithm. In the remainder of this section, we refer to SA-DYN as SA. We use RT and LR to refer to the ML-guided algorithms that employ regression tree (Algorithm 4.4) and logistic regression (Algorithm 4.5), respectively. We use RS for adaptive random search.

Setting. We apply the four algorithms (i.e., SA, RT, LR, and RS) to the 16 systems in Table 4.2. For each CI system, we execute the four algorithms for an equal time budget and then compare the results. For the non-CI systems, however, fixing the time budget may favour RS, as the other three algorithms have an additional overhead for training ML models. This overhead time is negligible when compared to the execution time for CI systems. But, for non-CI systems, we can execute many tests within the overhead time, which can skew the results in favour of RS. Therefore, to ensure that the results are valid for CI systems, we follow the approach proposed by Menghi et al. [MNBP20]. Specifically, we use the execution time of CI systems to limit the number of test inputs that each algorithm executes for non-CI objects. Briefly, to compare two algorithms with different overhead times, we allow the algorithm with the lower overhead time to execute x additional test inputs such that x multiplied by the execution time of a typical CI system (instead of a non-CI system) is equal to the difference in the two algorithms' overhead time. For the non-CI Simulink systems in Table 4.2, we use the average execution time of the Simulink CI system, i.e., autopilot. Given a time budget, we compute the maximum number of test executions that each of the SA, LR, RT, and RS algorithms can perform

within this time budget for autopilot. We then use these numbers to cap the number of test executions for each algorithm when we compare them for the non-CI models in Table 4.2. The time budget we consider for comparing these algorithms for CI systems and the maximum number of test executions we use to compare them for non-CI systems are available in our supplementary material [tab25e, tab25f]. We repeat each algorithm twenty times for each system except NTSS. For NTSS, due to its large execution time, we repeat each algorithm only ten times.

Metrics. We use the datasets created by SA, LR, RT, and RS to build decision rule models (DRM). For hyperparameter tuning, we use Bayesian optimization [SLA12]. To avoid bias towards any particular algorithm, we use for tuning the union of the datasets obtained from our four algorithms. Having fixed the hyperparameters, we train a DRM separately for each dataset obtained from each repetition of our four algorithms. We evaluate the DRMs using three metrics: *accuracy*, *precision* and *recall*. Accuracy is the number of correctly predicted tests over the total number of tests. Since DRMs are mainly used to predict the failure class, we compute precision and recall for the failure class as follows: Precision is the number of fail-class predictions that actually belong to the fail class, and recall is the number of fail-class predictions out of all the actual failed tests in the dataset. We use randomly generated test inputs within the variables' default ranges to measure the accuracy, precision and recall of each DRM.

Features for learning. For the Simulink systems, we use as features the individual input variables of each system but exclude the following two kinds of variables: (1) Variables that are explicitly fixed to a value in a requirement (e.g., variable APEng discussed in Section 4.3.3). (2) Reference variables that indicate the desired value of a controlled

process, noting that the system is expected to satisfy its requirements for *any* value in a reference variable's valid range. As such, reference variables cannot contribute to creating failure conditions. The desired altitude variable discussed in Section 4.3.3 is an example of a reference variable.

For NTSS, as discussed in Section 4.3.3, we consider alternative features as follows: the set of all individual variables (i.e., individual NTSS classes), sums of two variables, sums of three variables, ..., and the sum of all eight variables. We then create, for each input feature, one DRM based on a dataset obtained by each of the four algorithms. In total, for each algorithm, we create 248 DRMs for NTSS. That is, the sets of all subsets larger than two (247) and the set of all individual variables. Given the large number of hypothesized input features, we evaluate the accuracy of the resulting DRMs and keep the input features that yield reasonably high accuracy over multiple runs of SA, LR, RT, RS. This results in the retention of two input features for NTSS with an accuracy higher than 80%.

Results. Figures 4.8(a)-(c) compare across all the 16 systems the average accuracy of DRMs obtained by SA, LR and RT (on y-axis) against the average accuracy of DRMs obtained by RS (on x-axis). Each point in each of Figures 4.8(a)-(c) corresponds to one case-study system. A blue point indicates that the difference in accuracy is statistically significant as per the Wilcoxon rank-sum test. The DRMs obtained using SA are significantly more accurate than those obtained using RS for 14 out of 16 systems, including all the CI systems. The accuracy of the DRMs obtained using LR is significantly better than that obtained using RS for seven systems. The accuracy of the DRMs obtained using RT is significantly better than that obtained using RS for nine systems. Overall for all the systems, SA has the highest average accuracy (83%), followed by RT and LR with average

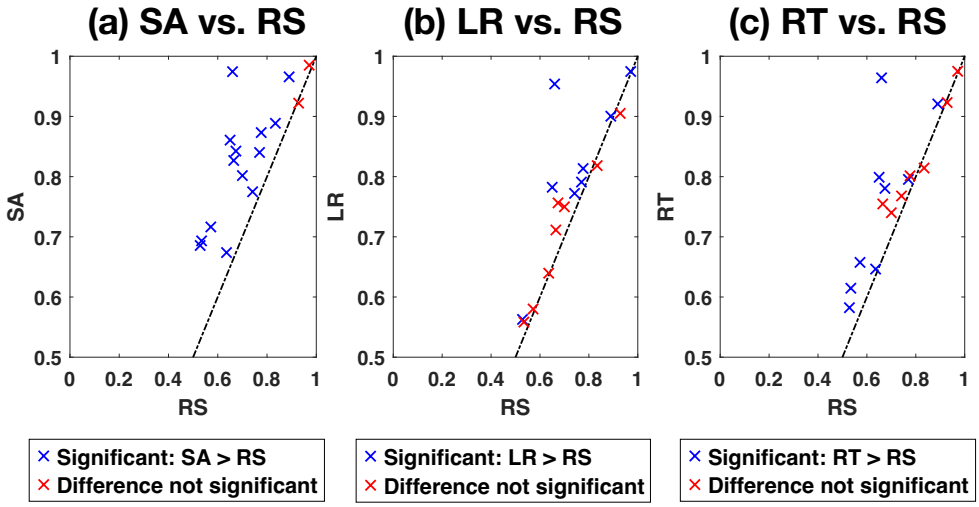


Figure 4.8: Comparing the accuracy of decision-rule models obtained based on the datasets generated by SA, LR, and RT against those obtained by RS for all the 16 systems in Table 4.2.

accuracies of 78% and 76%, respectively. RS has the lowest average accuracy (72%). Finally, the accuracy of SA, RT and LR is significantly better than that of RS. The effect size for SA versus RS is medium, and the effect size for RT and LR versus RS is small.

Figure 4.9 compares the average accuracy of the DRMs obtained using SA, LR, RT and RS for our 16 systems as the execution-time budget is varied from 50% to 100%. Note that in our experiments, we dedicate 50% of the time budget to the prepossessing step. Therefore, Figure 4.9 compares the impact of the four algorithms over the remaining 50% of the execution-time budget. As the figure shows, SA consistently has the highest average accuracy. Further, the average accuracy of RS reaches a plateau, whereas the other three algorithms keep improving as the budget increases. The main reason for this difference is that RS, unlike the other algorithms, does not use ML models to guide its test input sampling. Specifically, RS generates test inputs randomly across the entire input space.

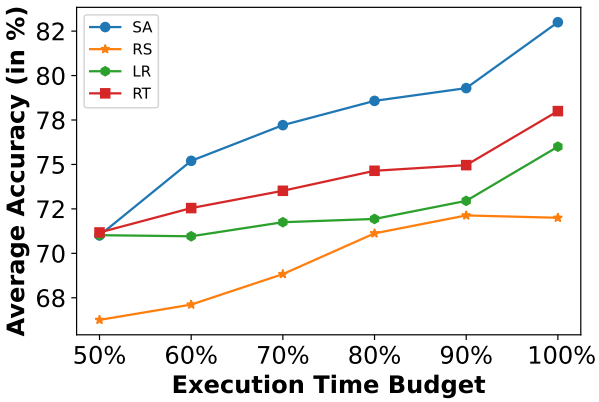


Figure 4.9: Average accuracy of the DRMs obtained using SA, RL, RT and RS for the 16 systems as the time budget is varied.

In contrast, RT and LR exploit boundary regions that separate passing and failing test inputs, resulting in a steady increase in accuracy as the search budget grows. Further, SA utilizes surrogate models, generating significantly more test inputs compared to the other algorithms within the same allotted time budget. As a result, SA achieves the highest average accuracy among all the algorithms.

Figure 4.10 compares the recall and precision for all the DRMs obtained using SA, LR, RT and RS for our 16 systems. Recall measures the ability of DRMs to precisely identify the failure conditions, whereas precision assesses the ability of DRMs to generate failure instances correctly. The SA algorithm has the highest average recall (88%), followed by LR (87%) and RT (85%). RS has the lowest average recall (83%). Moreover, SA has the highest average precision (72%), followed by RT (64%) and LR (62%) and RS has the lowest average precision (57%). The recall of SA, RT and LR is significantly better than that of RS with a medium effect size for SA, a small effect size for LR, and negligible effect size for RT. Likewise, the precision of SA, RT and LR is significantly better than that of RS with a

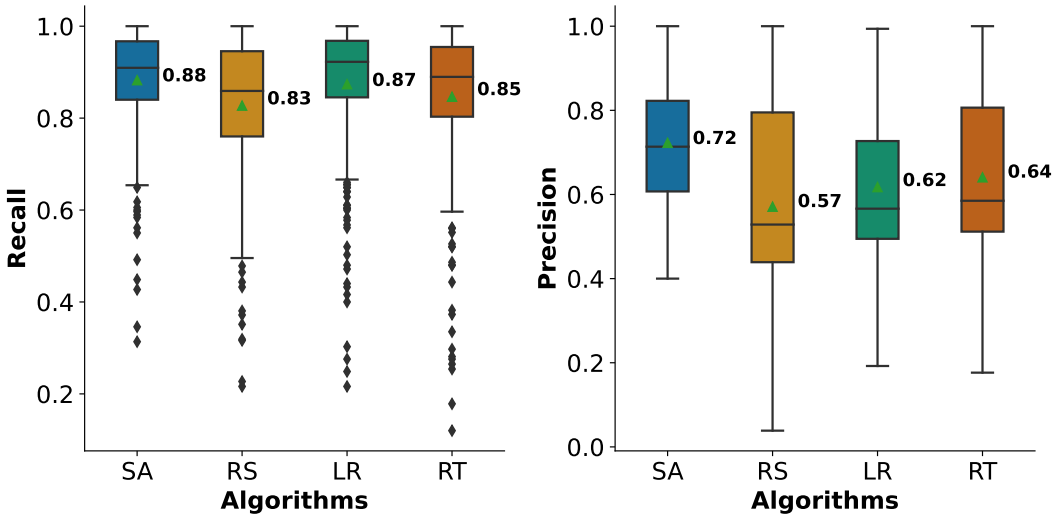


Figure 4.10: Recall and Precision for the DRMs obtained based on SA, RL, RT and RS for all the 16 systems in Table 4.2.

medium effect size for SA, small effect size for RT and negligible effect size for LR. Finally, the accuracy, recall and precision values for SA are significantly higher than those for RT and LR. Precision and recall for SA exhibit similar trends to that for accuracy (Figure 4.9). Detailed charts comparing precision and recall for the four algorithms as the execution-time budget varies are available online [fig25a, tab25g, tab25a].

Finding. Our evaluation for RQ2 performed over all our case-study systems shows that our dynamic surrogate-assisted approach yields failure models with significantly higher accuracy, precision and recall compared to those obtained using ML-guided algorithms and a random baseline.

4.4.4 RQ3-SOTA Comparison

We compare SA – the top-performing algorithm from RQ2 – with an adaptation of Alhazen to numeric-input systems. We hereafter use SoTA to refer to this adaptation, discussed next. Similar to SA, in SoTA, we generate test inputs according to the description in Section 4.4.1. The workflow of SoTA then matches the steps in Figure 4.1 with the difference that the model trained and refined in the main loop (i.e. step 3) is always a decision tree model; this decision tree is returned as the failure model at the end. Recall that our approach has a separate step, i.e., step 5, after the main loop to build failure models from the data generated by different algorithms. This step is not required in SoTA, noting that the model that SoTA refines during its main loop is used as the failure model. Algorithm 4.6 shows our implementation of SoTA. SoTA starts from an initial dataset (line 1). In each iteration, it builds a decision tree on the dataset (line 3). It then generates test inputs using all the paths in the decision tree (lines 4-10). These test inputs are executed and added to the dataset along with their labels (line 11). The final decision tree is returned on line 14.

Setting. For this comparison, we apply SoTA to our CI-systems in Table 4.2, i.e., NTSS, AP1, AP2 and AP3. For the decision tree parameters, e.g. maximum depth of tree and class weight, we use the same parameters as in the original study [KHSZ20, alh25d]. We execute SoTA for the same time budget as SA. For SA, we use the dataset generated in RQ2. We repeat SoTA for twenty times for each system except for NTSS. For NTSS, we repeat it only ten times due to the expensive execution time.

Metrics. In order to compare SoTA and SA, we build decision trees based on the datasets generated by SA in RQ2. To do so, we use the same decision tree parameters as those used

Algorithm 4.6 Our implementation of SoTA

Input S: System
Input $\mathcal{R} = \{R_1, \ldots, R_n\}$: Ranges for Input variables v_1 to v_n
Input F: Fitness Function
Input InitialDatasetSize: size of initial dataset
Output *DecisionTree*: A decision tree model (failure model)

1: $DS \leftarrow$ Preprocessing(S, $\mathcal{R}$, F, InitialDatasetSize); //Run Algorithm 1 to generate an initial dataset
2: **while** (execution budget remains) **do**
3: *DecisionTree* $\leftarrow$ Train(DS);
4: Let P_1,..., P_q be all the paths obtained from *DecisionTree*; // Extract all the paths from the decision tree
5: **for** each path P_k s.t. $k \in \{1, \ldots, q\}$ **do** // Generate a test in each path based on the ranges obtained from that path
6: Let R'_{i_1},..., R'_{i_m} be reduced ranges obtained from P_k;
7: **for** each variable v_{i_j} s.t. $j \in \{1, \ldots, m\}$ **do**
8: $\mathcal{R} \leftarrow (\mathcal{R} \setminus \{R_{i_j}\}) \cup \{R'_{i_j}\}$; // Replace the range R_{i_j} of v_{i_j} in $\mathcal{R}$ with the new reduced range R'_{i_j} from line 6
9: **end**
10: $\{t\} \leftarrow$ GenerateTests($\mathcal{R}$, 1); // Generate a test in path P_k
11: $DS \leftarrow DS \cup$ Execute(S, $\{t\}$, F); // Compute fitness for t and add to DS
12: **end**
13: **end**
14: **return** *DecisionTree*

by SoTA. We compare the decision trees using the three metrics explained in RQ2, i.e. accuracy, precision for fail class and recall for fail class. We also use the same test set utilized in RQ2.

Results. Figure 4.11 compares the average accuracy of the decision trees obtained by SA (on y-axis) against those obtained by SoTA (on x-axis) across the four CI systems in Table 4.2. Similar to Figure 4.8, each point on Figure 4.11 corresponds to one case-study system and a blue point denotes a statistically significant difference in accuracy, determined using the Wilcoxon rank-sum test. As figure 4.11 shows, the decision trees obtained using SA are significantly more accurate than those obtained by SoTA, for three out of the four

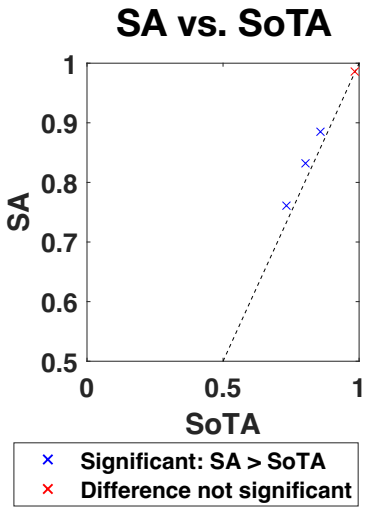


Figure 4.11: Comparing the accuracies of the decision trees obtained on the datasets generated by SA and the decision trees returned by SoTA for all the four CI systems in Table 4.2.

systems with a medium effect size. For the fourth system there is no statistically significant difference.

Figure 4.12 compares the average accuracy of the decision trees obtained using SA and those obtained by SoTA for the four CI systems as the execution-time budget is varied from 50% to 100%. As the figure shows, the average accuracy of SA is consistently higher than SoTA as the time budget increases.

Finally, Figure 4.13 compares the recall and precision for all the decision trees obtained using SA and those obtained by SoTA for the four systems. As shown by the figure, the average recall of SA (85%) is higher than SoTA (83%). Further, the recall of SA is significantly better than SoTA with small effect size. Moreover, the average precision of SA (76%) is

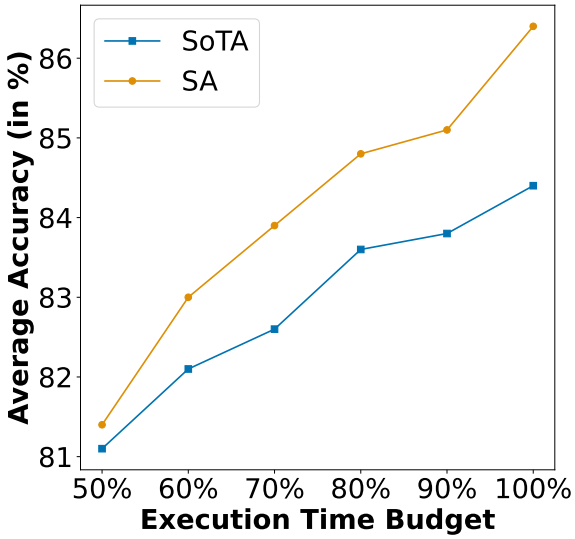


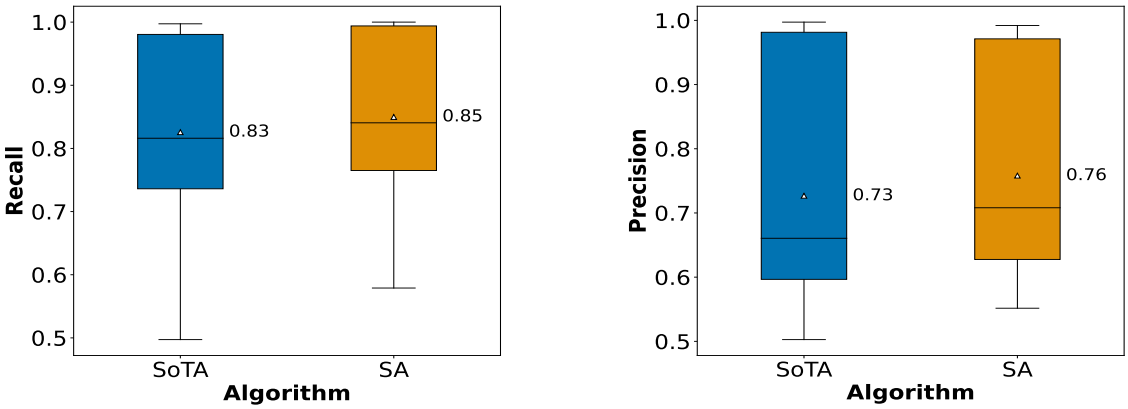
Figure 4.12: Average accuracy of the decision trees obtained using SA and SoTA for the four CI systems as the time budget is varied.

higher than SoTA (73%). Similar to recall, the precision of SA is significantly better than SoTA with small effect size.

Finding. Our evaluation for RQ3 performed using dynamic surrogate-assisted and the state-of-the-art approach over CI systems indicates that our dynamic surrogate-assisted approach yields failure models with higher accuracy, precision and recall compared to those obtained from the state-of-the-art approach.

4.4.5 RQ4-Usefulness

In view of the results of RQ2, we use the DRMs generated by the SA algorithm to evaluate their usefulness in identifying spurious failures. We focus on the DRMs for the CI systems in



(a) Recall (b) Precision

Figure 4.13: Recall and Precision for the decision trees obtained based on SA and SoTA for four systems in Table 4.2.

Table 4.2, i.e., NTSS, AP1, AP2 and AP3. For these systems, we can validate whether the inferred rules lead to genuinely spurious failures. For NTSS, we had access to an expert from industry, and for autopilot, we had detailed requirements and design documents. Recall from Section 4.3.3 that the rules we obtain from DRMs are in the form of *IF-condition-THEN-prediction*. Each rule has a confidence that shows what percentage of the tests satisfying the condition of the rule conform to the rule's prediction label. From the DRMs for each of the four CI systems, we extract the rules that predict the fail class with a confidence of 100%. These rules are candidates for specifying spurious failures, since they identify conditions that lead to and only to failures. We then select the rules that are not subsumed by others through logical implication. We use the Z3 SMT solver [DMB08] to find logical implications. In the end, we obtain seven rules for NTSS, 17 rules for AP1, 24 rules for AP2 and 15 rules for AP3. On average, the rules for NTSS include two variables and three predicates, and the rules for autopilot include three variables and four predicates.

To validate the rules for NTSS, we presented the rules to a domain expert. Our domain expert for NTSS is a seasoned network technologist and software engineer with more than 25 years of experience. The expert has been using the core enabling technology of NTSS (CAKE [HJTM18], discussed earlier in this section) in commercial networking solutions since 2018. Among the seven rules for NTSS, one rule constrains an input feature formulating a sum of the NTSS input variables. The rest of the rules involve predicates over individual input variables. The domain expert approved all the rules as they all correspond to situations where NTSS is overloaded with large traffic volumes, and hence, poor network quality is expected. We note that, for each input variable of NTSS, there is a threshold that is determined by the NTSS setup configuration. Among the seven rules, six constrain input variables near these known threshold values. An example of such rules is:

$\text{class6} \geq 91\% \cdot \text{thresh6} \wedge \text{class7} \geq 87\% \cdot \text{thresh7}$ THEN FAIL

where thresh6 and thresh7 are thresholds of class6 and class7, respectively. These rules matched the expert's intuition; nonetheless, the expert still found the rules helpful as they provide data-driven evidence for what the expert could only estimate based on experience and ad-hoc observations rather than systematically collected data. More importantly, the rule constraining a sum of input variables (as discussed in Section 4.2) was of particular interest. For this rule, neither could the domain expert estimate the combination of the variables in the rule nor was the limit in the rule close to known thresholds. This rule prompted an investigation into the source code of CAKE. This investigation confirmed that classes 7, 6 and 5 have higher priority than other classes. As such, the combined cumulative usage of these three classes needs to be capped to maintain high quality of

experience. The expert indicated that, without our approach, he would not have been able to formulate such a rule merely based on his existing test scenarios and expert knowledge.

For the autopilot's three requirements (AP1, AP2 and AP3) , we could conclusively confirm that 47 out of the 56 inferred rules represent spurious failures as per the handbook of the De Havilland Beaver aircraft [(AS09]. For the remaining nine rules, we could not ascertain whether they represent spurious failures. These rules either fail due to real faults in the system or they are spurious, but further expertise is required to confirm their spuriousness. The full set of rules and our analysis are in our supplementary material [tab25b].

Our results indicate that none of the 47 rules confirmed as inducing spurious failures for autopilot could be obtained solely from the datasets generated by the preprocessing phase of SA (see Figure 4.1). Similarly, only two of the seven rules for NTSS could be obtained using the datasets from the preprocessing phase. Furthermore, from the preprocessing phase datasets alone, no additional candidate rules could be inferred for spurious failures in either NTSS or autopilot. These findings highlight the importance of utilizing surrogate-assisted test generation in order to obtain useful and more comprehensive rules for spurious failures.

Finding. Our validation of failure-inducing rules against domain knowledge indicates that our dynamic surrogate-assisted approach is effective for identifying spurious failures. Indeed, our results show that expert judgement alone or tests generated without the assistance of surrogates would miss many rules that one would be able to identify using our proposed approach.

Data Availability. Implementations of all algorithms are available at [sou25]. The requirements specifications for all case-study systems are provided at [req25]. Our evaluation data includes: (1) raw datasets for the experiments [dat25]; (2) rules generated for CI systems [APN25, tab25b]; (3) evaluation scripts [eva25a] and the analyzed data [eva25b]; (4) statistical analysis results [sta25, tab25g, tab25d]; and (5) scripts for the plots in the chapter [eva25a].

4.4.6 Threats to Validity

The most important threats concerning the validity of our experiments are related to the internal and external validity.

4.4.6.1 Internal Validity

The *internal validity* risks are related to confounding factors. The effectiveness of failure-inducing rules inferred by our approach depends on the accuracy of fitness functions and the quality of the input datasets. For Simulink models, we use an automated and provably sound technique to obtain fitness functions from logical specifications [MNGB19]. However, the translation of natural-language requirements into logical specifications remains a manual task and necessitates domain-expert validation. In our experiments, we mitigated the risks associated with the accuracy of fitness functions as follows: For Simulink models, the fitness functions are automatically obtained from logical specifications approved by the engineers who developed the benchmark Simulink models [NGM+19]. For the NTSS case study, we validated the fitness function with our domain expert [JNSS23]. As for the

risks related to the quality of datasets, we note that the labels for the data points are computed based on the actual system outputs, and hence, are always accurate. Further, we used adaptive random testing in the preprocessing step (see Figure 4.1) to diversify the generation of the datasets in the search input space.

To address any concerns regarding our comparison with SoTA in RQ3, we conducted a comprehensive review of the SoTA code [alh25c] to ensure that the workflow of Algorithm 4.6, our adaptation of SoTA, conforms to the original SoTA code. In addition, we employed the same hyper-parameters for decision trees in Algorithm 4.6 as those used by SoTA. Finally, we have disclosed the code of Algorithm 4.6 in our GitHub repository [alh25a, alh25b] to facilitate further replication and comparison efforts.

4.4.6.2 External Validity

The systems we used for our evaluation and the characteristics of these systems may influence the generalizability of our results. Related to this threat, we note that: First, the Simuilnk models in Lockheed's benchmark represent realistic and representative CPS components from different domains. This benchmark has been previously used in the literature on testing CPS models [NGM+19, GPMS21]. Second, our case studies are drawn from two different domains: CPS and networks. Third, our network case study (NTSS) represents an industrial system for which we could interact with a domain expert. The above being said, our work would benefit from further experiments with a broader class of systems.

4.5 Lessons Learned

Lesson 1. Decision rules are a better choice than decision trees for building failure models. In this chapter, we focused on interpretable ML techniques for building failure models. Using these techniques, we were able to generate constraints on system inputs that are easily understandable to humans [Mol20]. Among interpretable ML techniques, decision rules and decision trees have been previously used in the literature for inferring rules pertaining to a specific behaviour of a system [BALS20, GMH15, HSNB21, GMN+20, KHSZ20]. We chose to use decision-rule models in our work for the following reasons: Decision rules are known to produce fewer and more concise rules compared to decision trees, which often generate many rules involving several variables and predicates. Further, decision trees are prone to the replicated subtree problem [WFH11]. This problem arises when the same subtree, with identical predicates and splits, appears multiple times in the tree. Replicated subtrees can increase model complexity, lead to overfitting, and hinder interpretability. Decision rules do not typically suffer from this problem, thus generally yielding more interpretable and less redundant rules. As mentioned in Section 4.4.5, the rules we obtain for NTSS and autopilot, on average, have three and four predicates over two and three variables, respectively. While one could argue that limiting the depth of a decision tree, as done by the SoTA baseline, would result in a reasonably small tree, our findings indicate that, decision trees built using the parameters of the SoTA baseline lead to a 40% higher number of rules and 10% more predicates compared to decision rules.

Lesson 2. To evaluate a test generation algorithm for systems with numeric inputs, the accuracy and usefulness of the failure models produced by the algorithm offer more

realistic insights about the algorithm than the number of individual failures found by the algorithm. For systems with numeric inputs, slight modifications to the inputs of a failure-revealing test may lead to redundant failures, i.e., failures caused by the same fault. Even when one considers input diversity, e.g., measured by the Euclidean distance between test-input vectors, one cannot determine whether failures are non-redundant or valid by merely analyzing individual test inputs. Consequently, evaluating testing algorithms solely based on their ability to generate failures may result in misleading conclusions. Indeed, had we premised our evaluation on the number of detected failures, we would have inferred that our dynamic surrogate-assisted algorithm produces 2.3 times more failures compared to the ML-guided and the baseline algorithms. While this conclusion would strongly favour our approach, we do not believe that this large margin is an accurate representation of the degree of improvement that our algorithm delivers. Based on the results of RQ2 and RQ3, the dynamic surrogate-assisted algorithm, when compared to alternatives, leads to an accuracy improvement ranging from 2% to 14%.

4.6 Summary

In this chapter, we presented a data-driven framework for inferring failure models for systems with numeric inputs including cyber-physical and network systems. The framework employs existing surrogate-assisted and machine learning-guided (ML-guided) test generation techniques. We proposed a new dynamic surrogate-assisted algorithm that uses multiple surrogate models simultaneously during search, and dynamically selects the predictions from the most accurate model. We compared the accuracy of failure models obtained using

our dynamic surrogate-assisted approach against two ML-guided techniques as well as two baselines using 16 case-study systems from the cyber-physical and network domains. Our results, confirmed by statistical tests, show that the average accuracy, precision and recall of the dynamic surrogate-assisted approach are higher than those of the ML-guided test generation algorithms, and of the state-of-the-art and random-search baselines. Moreover, the rules inferred from the failure models built for our compute-intensive systems identify genuine spurious failures as validated against domain knowledge.

Chapter 5

Learning Non-Robust Behaviours

In this chapter, we propose an approach for identifying conditions under which CPS exhibit volatility and non-robust behaviours (Challenge 4 in Section 1.2). We study this problem in the context of a network traffic shaping system (NTSS) as an industrial case study (Section 3.3), where identifying such non-robust behaviours is critical for ensuring stable network performance. We further introduce a robustness measure for NTSS that determines if a test input leads to an acceptable or unacceptable behaviour in the NTSS. In addition, we evaluate the accuracy of SOHOSim, our NTSS simulator discussed in Section 3.3, and provide insights into how it helps in revealing unknown and undocumented behaviours.

5.1 Introduction

Simulation-based testing has largely focused on discovering individual scenarios (tests) that can reveal system failures, e.g., system crashes or violations of some system requirement [MNBB14, BANBS16, FY20, Zel17]. While revealing system failures is an essential quality assurance task, simulators can be used for characterizing a system's non-robust behaviours. An input to a system reveals a non-robust behaviour when, by making small perturbations in the input, the output of the system changes from acceptable (passing) to unacceptable (failing) or vice versa [ALFS11]. For CPS, it is important to be able to identify test inputs that elicit non-robust behaviours.

In this chapter, we propose an approach that combines machine learning and adaptive random testing to generate value ranges for test inputs in response to which the system is likely to exhibit non-robustness. We apply our approach to our NTSS case study discussed in Section 3.3.

In collaboration with our industry partner, RabbitRun Technologies (https://www.rabbit.run), we develop a simulation environment to test NTSS. We then present Non-RobustnEss ANalysis for tRaffIC SHaping (ENRICH), a method to approximate input ranges that likely lead to non-robust NTSS behaviours. ENRICH implements an adaptive random testing algorithm based on our NTSS simulator. The test cases generated by adaptive random testing are used to train a regression tree from which the areas in the input search space that include system's non-robust behaviour are inferred. These areas are then passed to the adaptive testing algorithm to focus test generation in these inferred areas, since these areas likely include inputs that make the system non-robust. The iterative test

generation and regression tree model refinement continues until the computational budget is exhausted. The final regression tree will then be used to infer value ranges for the NTSS inputs that lead to non-robustness.

The regression tree model generated by ENRICH in the first iteration is trained on evenly distributed samples in the entire search space, thus yielding an explorative view. In contrast, the models generated in later iterations become more focused on inputs that likely make the system non-robust. These models provide an exploitative view on the desired regions of the search space. It is difficult to accurately approximate the whole search space relying on explorative views only. As shown in earlier research [MNBB14, WJD17], the gradual move from an explorative to an exploitative view, as adopted by ENRICH, is more effective at inferring promising areas of the search space, i.e., non-robust regions in this chapter.

We evaluate ENRICH on an NTSS setup recommended by RabbitRun. We compare ENRICH with a standard baseline based on random testing. The baseline infers non-robust test inputs using a regression tree model built based on samples uniformly selected from the search space (i.e., an explorative model without the gradual refinement into an exploitative model). Our results show that ENRICH significantly outperforms the baseline in generating and characterizing non-robust test inputs for NTSS. In particular, *ENRICH is able to identify non-robust test inputs with a precision of* 84% *and a recall of* 100% *while yielding a significantly higher overall accuracy than the baseline. In addition, our results show that there is no statistically significant difference between the test results obtained from our simulator and the results obtained by executing the same tests on a physical (hardware) testbed.*

Contributions. We make the following contributions:

(1) We introduce the problem of capturing non-robust test inputs for network traffic-shaping systems (Section 5.2).

(2) We present ENRICH – an approach to automatically infer input ranges that likely lead to non-robust NTSS behaviours (Section 5.3).

(3) We evaluate the accuracy of ENRICH and our NTSS simulator, SOHOSim (Section 5.4). Our experimental results are publicly available [enr25].

(4) We reflect on the lessons learned from our collaboration with RabbitRun Technologies (Section 5.5).

Structure. Section 5.2 motivates non-robustness analysis for NTSS. Section 5.3 presents our approach for characterizing non-robust test inputs. Section 5.4 describes our evaluation. Section 5.5 outlines our lessons learned. Section 5.6 summarizes the chapter.

5.2 Industrial Context and Motivation

RabbitRun Technologies (RRT) provides advanced network connectivity solutions for the SOHO market. RRT employs a well-established technique, known as traffic shaping [HJTM18], and develops a NTSS to support high-quality connectivity for SOHO, in particular for *real-time streaming* applications such as voice and video.

Figure 5.1 illustrates the working of an NTSS. If we transmit a voice message (e.g., "Hi, Can I talk to Mary?") without traffic shaping, the voice packets may be mixed with other

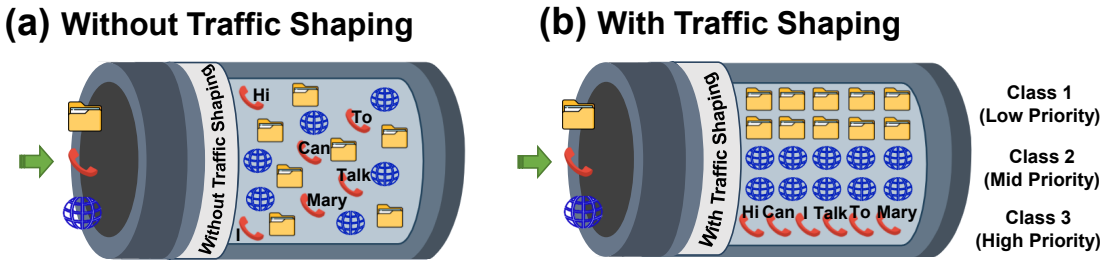


Figure 5.1: Illustration of network behaviour (a) without traffic shaping, and (b) with traffic shaping [tra].

packets. This makes it difficult to transmit the voice packets fast while keeping them in the same sequence they were sent (Figure 5.1(a)). Consequently, the receiver of the voice message, who needs to process the voice packets as they arrive, may feel that the voice is delayed or the conversation may be cut out for some seconds. To avoid the degradation of network quality for streaming applications, one can have an NTSS manage outbound traffic on routers. As shown in Figure 5.1(b), an NTSS divides the available bandwidth into a number of classes with different bandwidth thresholds and priorities. Voice and video packets are then allocated to a specific class where they can be transmitted together and in a timely manner.

NTSS can be set up with different numbers of classes and different rules for specifying bandwidth thresholds and priorities [HJTM18, cak]. The simple NTSS example in Figure 5.1(b) has three classes and is configured such that voice and video applications are allocated to `class3` – the highest-priority class. Browsing requests are mapped to `class2` with middle-level priority, and file sharing requests are mapped to `class1` – the lowest-priority class. For a complex NTSS setup, engineers need to have tools to help them configure the NTSS in the most optimal way. That is, they need to know which

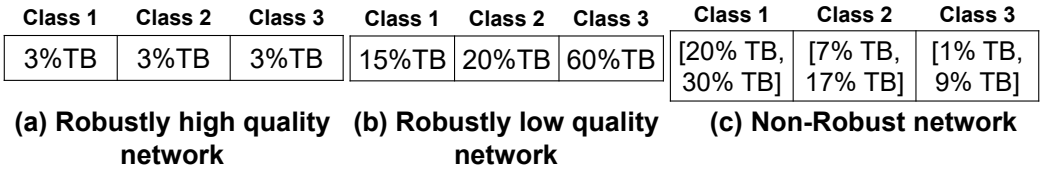


Figure 5.2: Input vectors for a network traffic-shaping system (NTSS) with three classes and a total available bandwidth denoted by TB: (a) low-bandwidth traffic (9% of TB) leading to a robustly high-quality network; (b) high-bandwidth traffic (95% of TB) leading to a robustly low-quality network; and (c) bandwidth ranges leading to a non-robust network.

applications should be mapped to which class to ensure the highest quality of service for streaming traffic without starving any other request types (e.g., browsing and file sharing).

To help engineers optimally configure an NTSS, we need the ability to approximate the boundary between robust and non-robust behaviours. Figure 5.2 exemplifies robust versus non-robust behaviours for an NTSS with three classes whose total available bandwidth is denoted by TB. In this system, for low outbound traffic (e.g., 9% of TB as shown in Figure 5.2(a)), users experience good-quality connection, and minor changes in the input do not affect the quality. On the other hand, for high outbound traffic (e.g., 95% of TB as shown in Figure 5.2(b)), users experience low-quality traffic that does not improve with minor changes in the input either. In other words, the examples in Figures 5.2(a) and (b) represent robust behaviours where for the former, the network quality is robustly good, and for the latter, the network quality is robustly bad. As one crosses between good-quality network situations to poor-quality ones, there is a sizable range of input traffic values where the system is volatile and where an input traffic stream with acceptable quality may turn bad due to small fluctuations in the input traffic ranges. Figures 5.2(c) illustrates example input ranges that may lead to *non-robust* behaviours in an NTSS.

Knowing about input ranges for which an NTSS likely becomes non-robust can help engineers in the following ways: The ranges can guide engineers in better mapping applications to classes. For example, if the engineers know that the non-robust range for `class1` is around 100mb/s and for `class2` around 150mb/s, they can assign applications to `class1` (resp. `class2`) with bandwidth values below 100mb/s (resp. 150mb/s). In this way, these ranges help solve the trade-off between optimal utilization versus providing acceptable and robust quality. Another use case for these input ranges is to devise run-time adaptation mechanisms that can steer the system away from non-robust regions, e.g., by dynamically reclassifying the traffic originating from different applications.

5.3 Approach

In this section, we present our approach for capturing non-robust test inputs in NTSS. Our approach leverages SOHOSim for test execution. Section 5.3.1 defines test inputs and outputs for NTSS. Section 5.3.2 introduces our robustness measure that enables us to distinguish between robust and non-robust behaviours. Section 5.3.3 presents Non-Robustn**E**ss A**N**alysis for t**R**aff**IC** S**H**aping (ENRICH) – our proposed approach for identifying non-robust test inputs in NTSS.

5.3.1 Test Input/Output Formalization

An NTSS consists of a set $C = \{c_1, \ldots, c_n\}$ of n classes. Each NTSS class c_i has a bandwidth range $[0..bwR_i]$ and a *priority*. We assume that the class indices represent their priority

order with c_n being the highest-priority and c_1 being the lowest-priority class. Let c_i and c_j be a pair of classes such that c_i has higher priority than c_j (i.e., $i > j$). A traffic-shaping algorithm provides better network quality (e.g., lower latency and loss) for the flows that go through c_i compared to those going through c_j as long as the bandwidth of the flows going through c_i (resp. c_j) remain below the maximum bandwidth of c_i (resp. c_j) [CAK25].

Each test input for an NTSS is a tuple $(tr_1, \ldots, tr_n)$ where each tr_i is the bandwidth of the flow going through class c_i. SOHOSim is able to generate flows with different bandwidths for different NTSS classes. The range for each tr_i is $[0..bwR_i]$. That is, we generate test inputs such that the flow in each class c_i remains below the maximum bandwidth of c_i.

A standard and well-known metric used in the network community to quantify network quality is *Mean Opinion Score (MOS)* [SWH16]. The MOS value is a real number ranging from 1.0 to 5.0, where 1.0 indicates the lowest quality and 5.0 indicates the best quality. We use MOS to measure network quality for each NTSS class. Specifically, for each test input $(tr_1, \ldots, tr_n)$, SOHOSim measures the MOS value corresponding to each input flow tr_i passing through class c_i. For example, suppose we test a four-class NTSS using a test input $(240, 230, 200, 100)$, and suppose SOHOSim measures MOS values 2.51, 3.3, 4.41, 4.49 for c_1, c_2, c_3 and c_4, respectively. That is, the tuple $(2.51, 3.3, 4.41, 4.49)$ is the output corresponding to the test input $(240, 230, 200, 100)$.

For each class c_i, engineers can determine a threshold for the MOS value measured for that class in order to differentiate between *good* (acceptable) network quality and *bad* (unacceptable) network quality. We refer to this threshold as the *MOS threshold* and

denote it by $mosTh$. The MOS threshold for each class can be determined based on domain knowledge and the configurations of an NTSS.

5.3.2 Robustness Measure

As discussed in section 5.3.1, for each test input $(tr_1, \ldots, tr_n)$, SOHOSim measures a tuple $(mos_1, \ldots, mos_n)$ as the test output such that each mos_i specifies the network quality for class c_i. In addition, for each c_i, we have a MOS threshold $mosTh_i$. If mos_i is higher than $mosTh_i$, the network quality at class c_i is acceptable (good); otherwise, the quality is unacceptable (low).

To be able to identify non-robust test inputs, it is not sufficient to determine the quality for each class individually. Instead, we need an aggregated measure that can determine, for a given test input, whether or not the NTSS performance as a whole (i.e., for all the classes) is acceptable. To do so, we need to combine the n MOS outputs obtained for a test input to compute a single measure. We adopt an approach that has been used in the search-based testing literature to define hybrid test objectives and combine several metrics simultaneously [FA12, McM04]. This approach allows one to not only aggregate different measures into one, but also to retain the priority of each measure in the aggregated measure. Specifically, for an NTSS, low quality of traffic on a higher-priority class is worse than low quality of traffic on a lower-priority class. We combine the MOS values measured for different NTSS classes in such a way that the aggregated measure preserves the priority of the classes. We refer to our single measure as *robustness measure*.

To define our robustness measure, we first normalize the MOS values obtained for each class. We use a well-known rational function $\omega(x) = x/(x+1)$ for normalization [Arc13]. We denote by $\overline{mos}_i$ the normalized form of each MOS value mos_i. For a given test output $(mos_1, \ldots, mos_n)$, we denote our robustness measure by $\mathcal{R}(\overline{mos}_1, \ldots, \overline{mos}_n)$ and define it as follows:

$$\begin{cases} \overline{mos}_n & \text{if } \bigwedge_{i=1..n} \overline{mos}_i < \overline{mosTh}_i, \\ 1 + \overline{mos}_{n-1} & \text{if } \overline{mos}_n \geq \overline{mosTh}_n \wedge \bigwedge_{i=1..n-1} \overline{mos}_i < \overline{mosTh}_i, \\ 2 + \overline{mos}_{n-2} & \text{if } \bigwedge_{i\in\{n-1,n\}} \overline{mos}_i \geq \overline{mosTh}_i \wedge \bigwedge_{i\in\{1..n-2\}} \overline{mos}_i < \overline{mosTh}_i, \\ \ldots & \ldots \\ n & \text{if } \bigwedge_{i=1..n} \overline{mos}_i \geq \overline{mosTh}_i \end{cases}$$

where $\overline{mosTh}_i$ is the normalized form of the MOS threshold $mosTh_i$. The robustness measure $\mathcal{R}$ is within the range $[0.5, n]$ since MOS values cannot go below 1.0 (see Section 5.3.1); as such, $\overline{mos}_i$ values cannot go below 0.5. A robustness value of 0.5 indicates that the network quality for the highest-priority class is low; a robustness value of n means that the network quality for all the classes is high. More precisely, the robustness measure is interpreted as follows:

$$
\begin{array}{lcl}
0.5 \leq \mathcal{R} < 1 & \Rightarrow & c_1 \ldots c_n \text{ are low quality} \\
1.5 \leq \mathcal{R} < 2 & \Rightarrow & c_1 \ldots c_{n-1} \text{ are low quality,} \\
 & & c_n \text{ is high quality} \\
\ldots & & \ldots \\
n - 1 + \frac{1}{2} \leq \mathcal{R} < n & \Rightarrow & c_1 \text{ is low quality,} \\
 & & c_2 \ldots c_n \text{ are high quality} \\
\mathcal{R} = n & \Rightarrow & c_1 \ldots c_n \text{ are high quality}
\end{array}
$$

If the robustness measure for test i is higher than that for test j, then the network quality is higher for test i than for test j. To differentiate between acceptable and unacceptable behaviours of an NTSS, engineers can set a threshold on the robustness measure; we denote this by $rbTh$.

The input flows of an NTSS constantly fluctuate. Hence, it is critical to be able to distinguish between robust and non-robust inputs. The closer the robustness measures of inputs to the robustness threshold $rbTh$, the more non-robust those inputs are. Figure 5.3 illustrates the range of our robustness measure and specifies the robust and non-robust parts within this range. Test inputs whose robustness measures are close to $rbTh$ are more likely to flip system behaviour from being acceptable to unacceptable (or vice versa) by a small change, e.g., a change of %1 in the flow bandwidths. Dually, inputs with robustness measures far away from $rbTh$ are unlikely to flip system behaviour due to minor perturbations in the input.

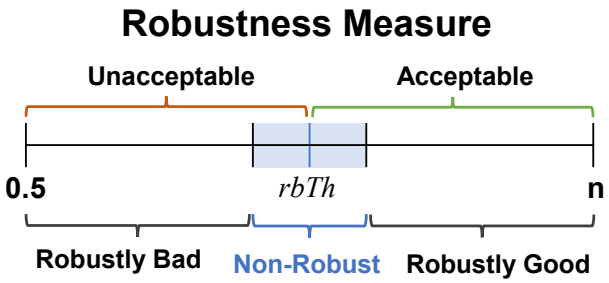


Figure 5.3: Illustrating the relationship between robust versus non-robust and acceptable versus unacceptable sub-ranges within the robustness measure range.

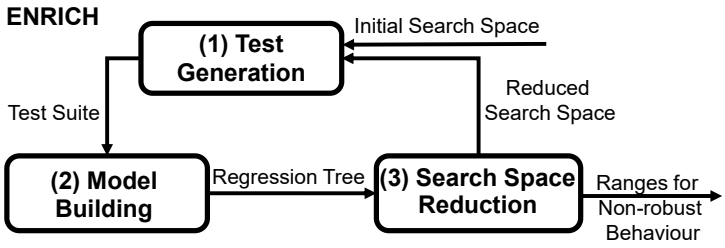


Figure 5.4: An Overview of ENRICH.

5.3.3 Non-Robust Behaviour Characterization

Figure 5.4 shows an overview of ENRICH – our approach for characterizing non-robust test inputs. ENRICH performs, in an iterative manner, the following tasks: (1) Generating a set of test inputs (TS) within a given search space. (2) Building a regression tree model (RT) using the test inputs TS generated in the previous step and their robustness measure outputs. (3) Using the regression tree model from step 2 to compute input ranges yielding non-robust outputs (i.e., robustness measures close to the robustness threshold $rbTh$). These ranges are used as the reduced search space in the next iteration. The process continues until the test budget runs out.

Figure 5.5 shows two regression trees generated by ENRICH in two consecutive iterations. The example is for an NTSS setup with four classes. Hence, each test input has four input variables tr_1 to tr_4. The initial search space for each tr_i is $[0..bwR_i]$. ENRICH constrains the range for each tr_i iteratively so that the range can capture values leading to non-robustness. In the first iteration (Figure 5.5(a)), ENRICH is able to constrain ranges for tr_1, tr_2 and tr_3, but the range for tr_4 is left unconstrained. Note that the ranges generated by ENRICH are parameterized in the form of $[v - \varepsilon, v + \varepsilon]$, where v is a value derived from the regression tree representing the boundary between acceptable and unacceptable values for the robustness measure. Later in this section, we describe in detail how the value v for a variable tr_i is derived from the regression tree. In the next iteration, for each variable tr_i that is already constrained, we generate tests by sampling tr_i in $[v-5\% \cdot TB, v+5\% \cdot TB]$. This is to ensure that ENRICH is exploiting the search region that is in the close proximity of the non-robustness threshold.

Figure 5.5(b) shows the regression tree generated in the second iteration and the ranges induced by this tree. In the second iteration, we are able to constrain tr_4, and refine the ranges for tr_1 and tr_3 into more precise ranges. We note that the new ranges for tr_1 and tr_3 are not too far away from their ranges in the previous iteration. That is, the search for the input ranges tends to exploit specific areas in the search space. We further note that in the second iteration, the tree does not produce any ranges for tr_2. In this case, ENRICH retains the range from the previous iteration, thus ensuring that input ranges are replaced only when a more precise range has been computed. In Section 5.4, we evaluate the accuracy of the final parameterized ranges generated by ENRICH to assess how well

these ranges capture non-robustness in NTSS. In particular, we study the relationship between the size of the ranges (i.e., the value of ε) and the accuracy of ENRICH.

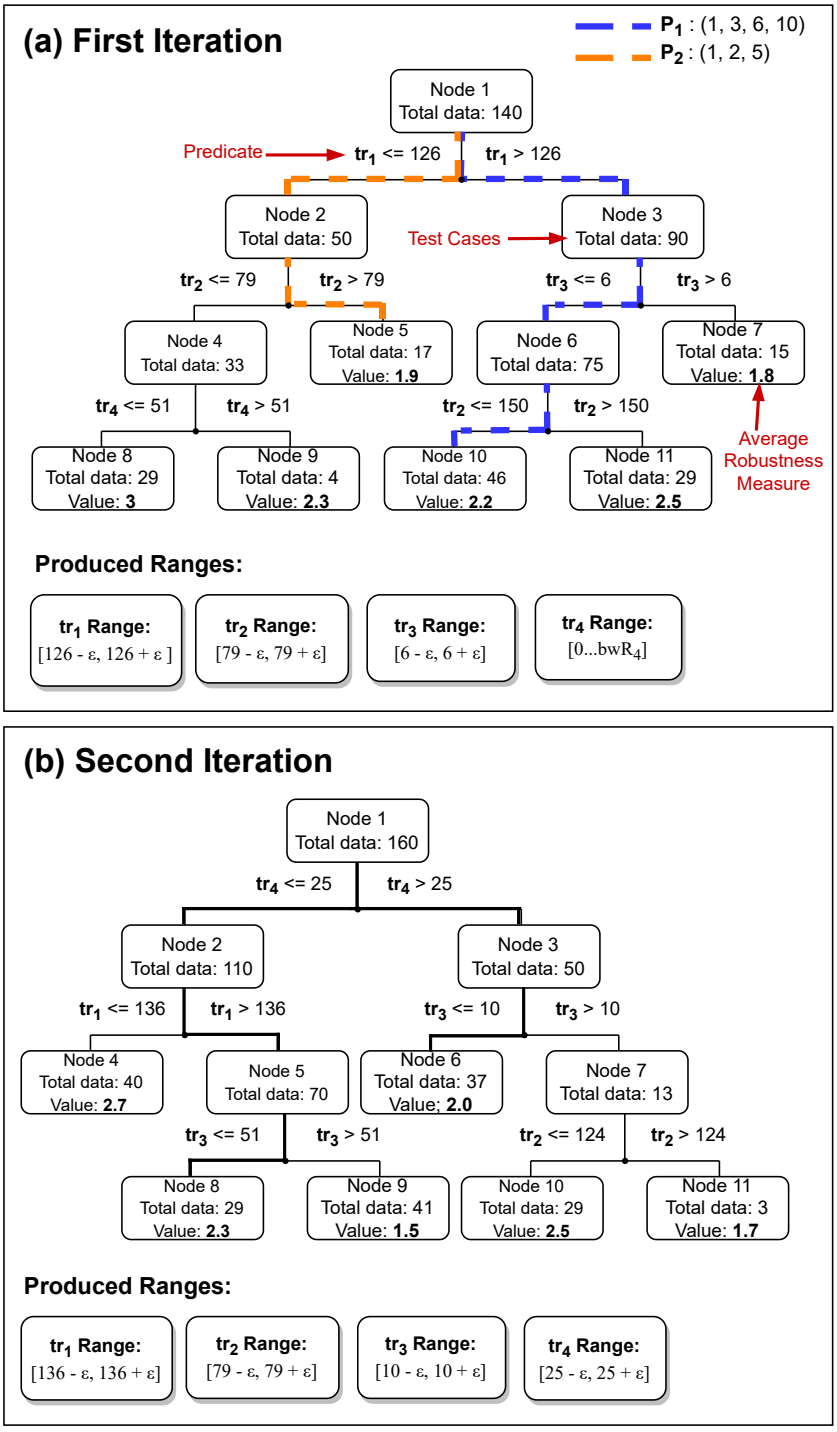


Figure 5.5: Illustration of two successive iterations of ENRICH and the induced input ranges.

Algorithm 5.1 The ENRICH Approach.

Input *Sys*: Executable system or simulator
Input $(R_1, \ldots, R_n)$: The initial search space
Output NR: Input ranges characterizing non-robustness

1: TS = TSAll = $\emptyset$; NR = $(R_1, \ldots, R_n)$; //Variables Initialization
2: **for** (i = 0 to BUDGET) **do:**
3: TS = GENTESTS(*Sys*, NR) //Test input generation
4: TSAll = TS $\cup$ TSAll; //Combine new and old tests
5: *RT* = BUILDRT(TSAll);//Build regression tree
6: NR = REDUCE(*RT*); //Search space reduction
7: **end for**;
8: **return** NR;

ENRICH is implemented by Algorithm 5.1. The input of ENRICH is a simulator, *Sys* (e.g., SOHOSim), and input ranges $R_1, \ldots, R_n$ (i.e., $[0..bwR_1] \times \ldots \times [0..bwR_n]$). ENRICH uses an adaptive random testing (ART) [Luk13] algorithm to generate a set *TS* of test cases (Line 3). ART randomly samples tests from the search space by maximizing the distance between newly selected vectors and the previously generated ones, hence ensuring that the tests are evenly spread over the search space. ENRICH selects each test case in *TS* from the given input space $(R_1, \ldots, R_n)$ and executes it to compute its robustness measure.

ENRICH builds a regression tree model using the test suite built so far (Lines 4-5). See Figure 5.5 for examples of regression trees built by ENRICH. A regression tree recursively splits data at each inner node into two children by minimizing the sum of the squared deviations from the mean in each child node [reg]. The value of each leaf node represents the average of the robustness measures of the test cases in that node. The tree edges are

labelled with predicates in the form of $tr_i \sim v$ where tr_i is an input variable, $v \in \mathbb{R}$ and $\sim \in \{\leq, >\}$.

Finally, ENRICH uses the regression tree model to constrain the ranges of the input variables (Line 6). A description of the REDUCE() routine is available in our supplementary material [alg25]. Briefly, ENRICH finds the two paths whose leaf-node values, i.e., the average of the robustness measure, are closest to the robustness measure threshold ($rbTh$). In Figure 5.5(a), assuming that $rbTh = 2.1$, we select $P_1 : (1, 3, 6, 10)$ and $P_2 : (1, 2, 5)$. We identify constraints induced by these paths such that the constraints only characterize test cases with robustness measures close to the threshold. In Figure 5.5(a), the constraint induced by P_1 (the path highlighted in blue) is $(tr_1 > 126) \wedge (tr_3 \leq 6) \wedge (tr_2 \leq 150)$, and the constraint induced by P_2 (the path highlighted in orange) is $(tr_1 \leq 126) \wedge (tr_2 > 79)$. The predicate $tr_2 \leq 150$ is not contributing towards identifying non-robust behaviours because its related branch at node 6 splits node 10 and 11 whose values are both above $rbTh$. Therefore, we eliminate the predicate $tr_2 \leq 150$. Specifically, we keep the predicates that lead to two nodes where one is above and one is below $rbTh$, and remove the rest as they do not help with the characterization of non-robust behaviours. We then simplify the constraints so that for each variable tr_i, we obtain at most one upper-bound predicate $(tr_i \leq v)$ and one lower-bound predicate $(tr_i > v)$, since, for example, any two predicates $(tr_i \leq v_1)$ and $(tr_i \leq v_2)$ such that $(v_1 \leq v_2)$ can be replaced by $(tr_i \leq v_1)$. At the end, for each predicate $tr_i \sim v$ where $\sim \in \{\leq, >\}$, we create a (parameterized) range $[max(v - \varepsilon, 0), min(v + \varepsilon, bwR_i)]$ where bwR_i is the maximum value that tr_i can assume. For each tr_i, its new range is passed to the test generation routine (Line 3). In the test generation routine, each tr_i for which a range $[max(v - \varepsilon, 0), min(v + \varepsilon, bwR_i)]$ exists is

sampled close to v (within $v \pm 5\% \cdot TB$). If tr_i is not constrained, then its range obtained in the previous iteration will be retained for the next iteration. For each variable tr_i, ENRICH ensures that the default range of $[0..bwR_i]$ will not be passed to the next iteration if the variable's range has been narrowed at some point. This is essential as the ultimate goal of the algorithm is to get narrower ranges for each tr_i.

5.4 Evaluation

We evaluate ENRICH and SOHOSim using two RQs:

RQ1. (Accuracy of ENRICH) *Do the ranges generated by ENRICH accurately capture non-robust behaviours of an NTSS?* We develop a set of tests labelled as robust and non-robust. We then use the ranges produced by ENRICH to predict the labels for these tests and assess the prediction accuracy of ENRICH. Since the ranges generated by ENRICH are parameterized (i.e., by the ε parameter), we evaluate the accuracy of ENRICH by varying ε and discuss how ε can be set in practice to address different needs: more accurate generation of non-robustness versus more accurate characterization (or coverage) of non-robustness. Further, since ENRICH is randomized, we study whether combining ranges obtained from multiple runs of ENRICH improves its accuracy.

RQ2. (Accuracy of SOHOSim) *Is there a significant difference between the test results obtained from SOHOSim (our simulator) and a hardware-in-the-loop testbed of NTSS?* Network testing is often performed using Hardware-in-the-Loop (HiL) testbeds [Him13]. SOHOSim, being a virtual testbed, provides flexibility and efficiency, allowing us to run

a large number of tests from which we can infer interpretable feedback (i.e., input ranges characterizing non-robustness). Nevertheless, as this RQ investigates, we need to ensure that the simulation results are close to those obtained over an actual NTSS that executes on hardware. To assess the accuracy of SOHOSim, we compare the results obtained from SOHOSim with the results obtained from a HiL-based NTSS testbed with identical configurations.

Data Availability. We have made the installation guidelines for SOHOSim, our implementations of ENRICH and BASELINE, and our experimental results publicly available [enr25].

5.4.1 RQ1-Accuracy of ENRICH

Before answering RQ1, we present the baseline, our experiments' parameters and setup, and the comparison metrics.

Baseline: To the best of our knowledge, there is no approach in the literature that performs what ENRICH does for network traffic-shaping systems. While there are approaches that develop interpretable ML from test results [ANBS18, GKH+20], none generate constraints in the form of ranges for input variables. To have a baseline, as per the empirical guidelines for search-based software engineering [gui21], we compare our approach with standard adaptive random testing (ART) and use the results to infer input ranges. In particular, our baseline (hereafter BASELINE) generates tests using ART within the default input ranges and then builds a regression tree using the test results only once. BASELINE uses the same parameters as ENRICH's, the main difference being that BASELINE is

Table 5.1: The parameters required by ENRICH

Parameter	Definition	Value
rbTh	The robustness measure threshold	3.6
TestSuiteSize	Number of test cases generated in each iteration of ENRICH	20
NodeSize	Minimum number of tests at the leaves of a regression tree	1
TB	The NTSS total bandwidth	400

non-iterative. That is, it obtains (parameterized) input ranges based on the results of a fully explorative search and skips the combination of explorative and exploitative searches as utilized by ENRICH.

Parameters and setup: The experiment parameters are shown in Table 5.1. We configure SOHOSim according to an industrial setup of NTSS recommended by RabbitRun. We refer to this setup as *NTSS-RR*. This setup uses an *8-tier mode* of CAKE known as diffserv8 [HJTM18], (i.e., $n = 8$). We use the default configuration of CAKE for the maximum bandwidth (bwR_i) of each class. We set the total bandwidth (*TB* in Table 5.1) to 400 Mbit as per the recommendation of our partner.

To answer RQ1, we apply ENRICH and BASELINE to NTSS-RR. We set the total number of tests generated and simulated by each run of ENRICH and BASELINE to 300. We arrived at this number based on preliminary experiments and setting a time budget of one day (give or take) for one run. In our setup, it takes approximately 27 hours, on average, for SOHOSim to perform 300 simulations.

For each run of ENRICH, in the first iteration, we generate 100 test inputs to have a sizeable number of data points for building an initial regression tree. We then perform 10 iterations of Algorithm 5.1 where we generate 20 test inputs in each iteration, thus

ensuring that the total number of simulations by ENRICH is 300. The stopping criterion (*NodeSize*) for regression tree creation is one, meaning that we expand the tree to the fullest extent. This makes it possible to derive constraints from the tree involving the most input variables, and hence, obtain more constrained input ranges. Based on feedback from our partner, we set the robustness threshold $rbTh$ to 3.6. We run ENRICH and BASELINE fifteen times each to account for random variation. Collectively, it took more than three weeks of computation to carry out all the runs. Performing more runs was not feasible due to time limitations. All the experiments were executed on a machine with a 2.5 GHz Intel Core i9-119900H CPU and 64 GB of DDR4 memory.

Comparison Metrics. We generate a set *TestSet* of 200 test inputs randomly and label them as robust and non-robust using the following procedure: For each test $tc \in$ *TestSet*, if the calculated robustness measure is below (resp. above) $rbTh$, we deduct (resp. add) a small perturbation over a short time period (around 2% of TB) from (resp. to) each input value in tc and simulate the resulting test to check whether the robustness measure has moved from below $rbTh$ to above it, or vice versa. A test input is labelled robust, if its robustness measure does not move from below $rbTh$ to above it, or vice versa. Otherwise, the test input is labelled as non-robust. Labelling test inputs as robust and non-robust is expensive since we need to run each test multiple times. It took more than two days to generate a labelled set of 230 test inputs. The perturbation size (2%) is based on the recommendations of our partner.

We label the tests in *TestSet* based on the input ranges that ENRICH and BASELINE generate. For example, if one run of ENRICH (or BASELINE) generates ranges [180..280]

for tr_1 and [32..42] for tr_2, we label a test as non-robust if and only if the values of both tr_1 and tr_2 in that test fall in those ranges. Otherwise, we label the test as robust.

We use the accuracy metric to compare the prediction ability of ENRICH and BASELINE. As noted earlier, the ranges generated by ENRICH and BASELINE for each input variable are in the form of $[v - \varepsilon, v + \varepsilon]$. We thus need to assign a value to ε to use the ranges for labelling. The value of ε allows us to control whether the input ranges are good at accurately generating non-robustness or at covering (characterizing) non-robustness. Figure 5.6 is a schematic view of the distribution of robust and non-robust tests for NTSS when the value of an input variable tr_i changes in its default bandwidth range $[0..bwR_i]$. When the values of input variables are close to the lower or upper bounds of their ranges, the resulting tests are more likely to be robust (i.e., robustly good for the lower bound, and robustly bad for the upper bound). But, when the values of input variables are in the middle, the tests are more likely to be non-robust. The input ranges generated in our work, which often fall in the middle, can be used for two different use cases: (1) They can be used to precisely predict (generate) non-robustness (i.e., yielding high precision for non-robustness), or (2) They can be used to characterize (cover) non-robustness (i.e., yielding high recall for non-robustness). As shown in Figure 5.6, we expect the ranges obtained by setting ε to a small value (e.g., 5% of the default range) to be better at predicting non-robustness, and the wider ranges (e.g., $\varepsilon = 40\%$) to be better at covering (characterizing) non-robustness. Hence, in addition to accuracy, we report for smaller ranges the *precision* for non-robustness, and for the wider ranges the *recall* for non-robustness. Full precision and recall results for both robustness and non-robustness are available online [enr25].

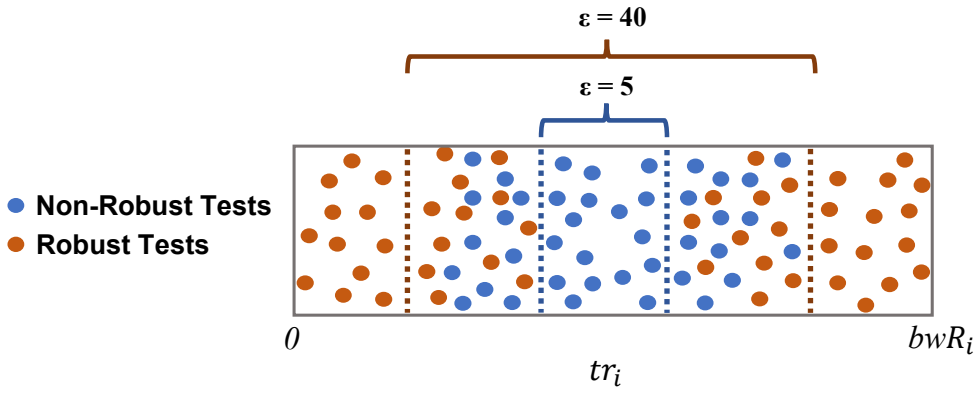
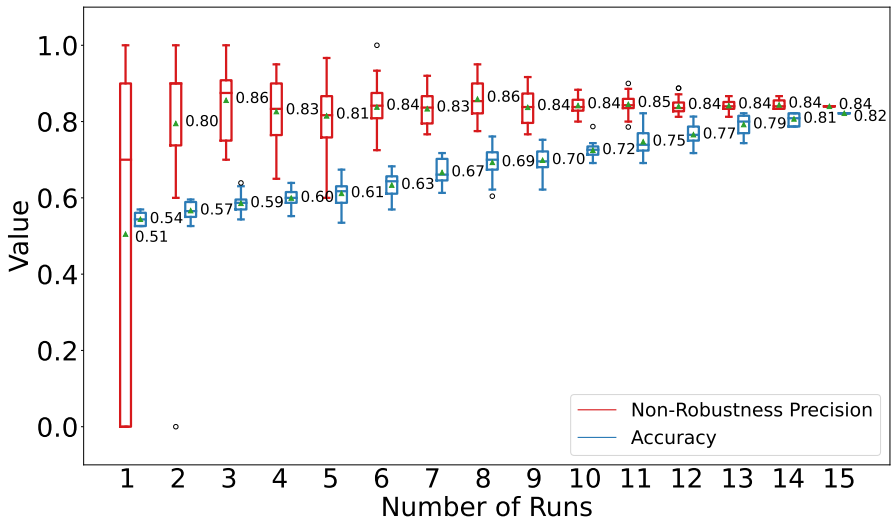


Figure 5.6: Distribution of robust and non-robust NTSS tests with respect to the default range of an input variable tr_i. Smaller ranges (lower ε values) are more precise in predicting non-robustness, and larger ranges (higher ε values) provide more coverage for characterizing non-robustness.

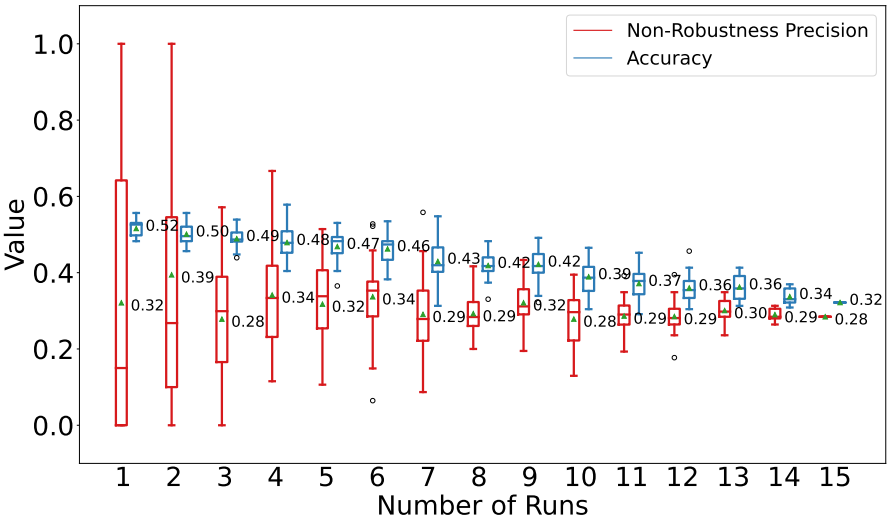
Since ENRICH and BASELINE are randomized, they likely generate different results when they are executed multiple times. We expect ENRICH, but not BASELINE, to generate overlapping ranges over multiple runs and exploit specific sub-ranges of the default input ranges instead of exploring the entire default ranges. To assess this difference between ENRICH and BASELINE, we compare ENRICH and BASELINE based on both their single runs and combinations of their multiple runs. Specifically, provided with n runs of ENRICH (resp. BASELINE), we label a test in *TestSet* as non-robust if and only if at least one of the n runs label the test as non-robust. Otherwise, we label the test as robust. We vary n from 1 to 15 (i.e., the maximum number of runs we have for both techniques). To account for randomness in selecting multiple runs, we randomly select 20 different combinations for $n = 2, \ldots, 13$. For $n = 1, 14$, we consider all the 15 possible combinations ($\binom{15}{1}$ and $\binom{15}{14}$); and, for $n = 15$, we consider all the runs together ($\binom{15}{15}$).

Results. To answer RQ1, we show the results for two cases: (1) Input ranges are used to generate non-robustness when we set $\varepsilon = 5\%$ to have narrow ranges, and (2) Input ranges are used to characterize (cover) non-robustness when we set $\varepsilon = 25\%$ to 40% to have wide ranges. As noted earlier, for both cases, we report accuracy. In addition, we report the precision for the non-robust class to assess the generation of non-robust behaviours (case-1), and the recall for the non-robust class to assess the coverage of non-robust behaviours (case-2).

Figure 5.7 shows the results for the first case which include the accuracy and non-robustness precision obtained from the combinations of n random runs of ENRICH and BASELINE, where we vary n from 1 to 15. As shown in the figure, the average accuracy and non-robustness precision of ENRICH are always higher than those of BASELINE. As we consider more run combinations, both the accuracy and the non-robustness precision of ENRICH increase significantly (from an average of 54% to 82% for accuracy, and from 51% to 84% for non-robustness precision). For BASELINE, however, both accuracy and non-robustness precision decrease considerably (from an average of 52% to 32% and from an average of 32% to 28%, respectively) as we consider more run combinations. Overall, using ENRICH, we are able to obtain an average accuracy of 82% and an average non-robustness precision of 84% when we consider all the runs. In contrast, the best averages for accuracy and non-robustness precision we obtain using BASELINE are 52% and 39%, respectively; these results are significantly lower than ENRICH's.



(a) Accuracy and non-robustness precision for ENRICH.



(b) Accuracy and non-robustness precision for BASELINE.

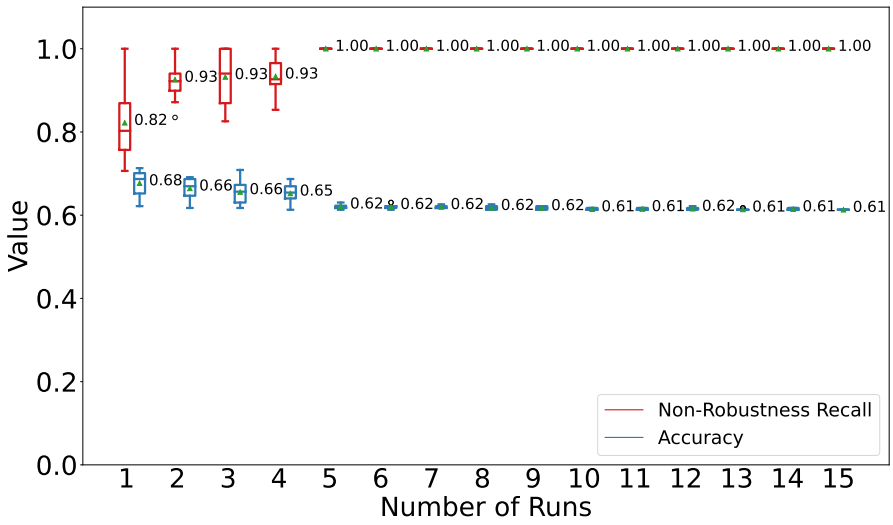
Figure 5.7: Evaluating the non-robustness generation use case ($\varepsilon = 5\%$) by comparing accuracy and non-robustness precision for different run combinations of ENRICH and BASELINE.

Table 5.2: Comparison of averages of accuracy and recall for the non-robustness class of ENRICH (E) and BASELINE (B) at $\varepsilon = 25\%, 30\%$ and 35%, and for their single runs as well as combinations of their 5, 10 and 15 runs.

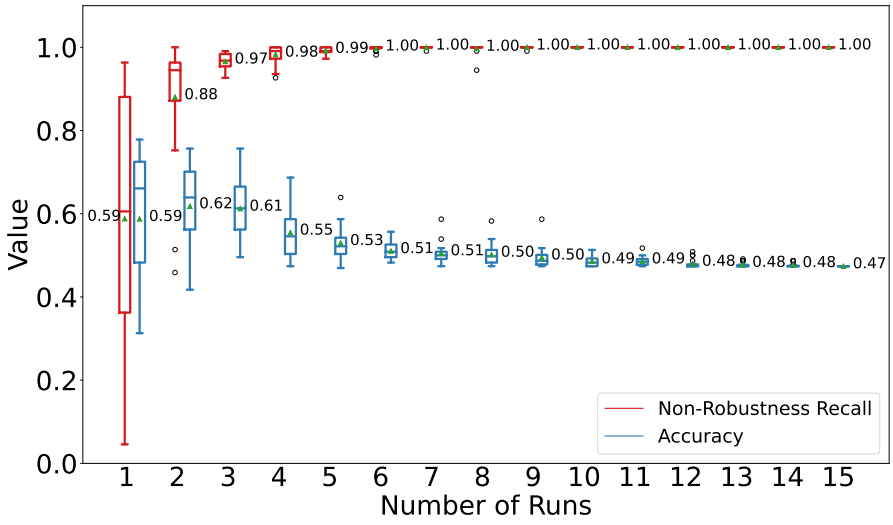
	1 ***Run***		5 ***Runs***	
	Accuracy(E - B)	**Non-Robustness Recall(E - B)**	**Accuracy(E - B)**	**Non-Robustness Recall(E - B)**
$\varepsilon = 25\%$	0.64 – 0.61	0.42 – 0.53	0.72 – 0.59	0.87 – 0.98
$\varepsilon = 30\%$	0.68 – 0.61	0.65 – 0.55	0.70 – 0.55	0.90 – 0.98
$\varepsilon = 35\%$	0.66 – 0.64	0.71 – 0.60	0.64 – 0.56	0.94 – 0.99

	10 ***Runs***		15 ***Runs***	
	Accuracy(E - B)	**Non-Robustness Recall(E - B)**	**Accuracy(E - B)**	**Non-Robustness Recall(E - B)**
$\varepsilon = 25\%$	0.72 – 0.52	0.93 – 1.00	0.71 – 0.47	0.95 – 1.00
$\varepsilon = 30\%$	0.69 – 0.49	0.94 – 1.00	0.68 – 0.47	0.97 – 1.00
$\varepsilon = 35\%$	0.63 – 0.50	0.96 – 1.00	0.62 – 0.47	0.98 – 1.00

Dually to Figure 5.7, Figure 5.8 compares ENRICH and BASELINE for the second case when we have $\varepsilon = 40\%$ of the default ranges. The figure compares the accuracy and non-robustness recall for the n run combinations of ENRICH and BASELINE where n varies from 1 to 15. Similar to the previous case, the average accuracy of ENRICH is always higher than that of BASELINE. The average non-robustness recall of ENRICH is higher than that of BASELINE for a single run and is almost the same for the other run combinations. Both approaches achieve almost 100% non-robustness recall for $n \geq 6$. However, when both approaches achieve full recall, ENRICH always maintains an average accuracy that is 10% or more higher than that of BASELINE. In addition, we show the averages for accuracy and non-robustness recall for $\varepsilon = 25\%$, 30% and 35% and for $n = 1, 5, 10, 15$ in Table 5.2. The full box-plots are available online [alg25]. The results in the table are consistent with those for $\varepsilon = 40\%$. Specifically, ENRICH is always more accurate than BASELINE. For the 10- and 15-run combinations where both approaches achieve a non-robustness recall of above 90%, ENRICH maintains an average accuracy that is 13% to 24% higher than that of BASELINE.



(a) Accuracy and Non-Robustness Recall of ENRICH.



(b) Accuracy and Non-Robustness Recall of BASELINE.

Figure 5.8: Non-robustness characterization use case ($\varepsilon = 40\%$); comparing accuracy and non-robustness recall for different run combinations of ENRICH and BASELINE.

Overall, the results show that the ranges generated by ENRICH, due to its exploitative search, tend to converge in a way that they can generate and characterize non-robustness with a higher accuracy compared to BASELINE. On the other hand, the relatively low accuracy of BASELINE indicates that the non-robustness recall results of BASELINE are more due to the random (explorative) nature of its search. Specifically, the high recall for non-robustness in Figure 5.8(b) is because BASELINE labels *all* the tests as non-robust. Hence, while it achieves high non-robustness recall, its accuracy is lower than 51%. This is confirmed by the detailed precision and recall results for both robustness and non-robustness that is available online [enr25].

Finding. The answer to RQ1 is that ENRICH consistently generates and characterizes non-robustness with a significantly higher accuracy than BASELINE. Specifically, ENRICH generates non-robustness with a precision of 84% while outperforming BASELINE in the overall accuracy by at least 30%. In addition, ENRICH characterizes non-robustness with 100% recall and yields an accuracy that is at least 10% higher than that of BASELINE.

5.4.2 RQ2-Accuracy of SOHOSim

To answer RQ2, we randomly generate 100 test inputs, and simulate them on SOHOSim and on a hardware-in-the-loop (HiL) testbed for NTSS where we replace the router and Internet modem in Figure 3.4 with their equivalent hardware components. We refer to the HiL testbed as SOHOHW.

Table 5.3: Mean and standard deviation for random tests obtained from SOHOSim and SOHOHW. Two environments are also compared based on p-value and MAE.

SOHOSim		***SOHOHW***		***Comparison***	
Mean	**Standard Deviation**	**Mean**	**Standard Deviation**	**P-value**	**MAE**
2.26	0.995	2.48	0.944	0.18	0.25

We compute the robustness measures for each test based on the results obtained from SOHOSim and SOHOHW. Table 5.3 shows the values of mean and standard deviation obtained for 100 test inputs executed on SOHOSim and SOHOHW. We have conducted Mann–Whitney U Test (level of significance is 0.05) on these two distributions. The p-value is 0.18 indicating that there is no significant difference between the two distributions. In addition, the mean absolute error (MAE) of the robustness measures of the 100 test inputs obtained from SOHOSim and SOHOHW is 0.25, which is about 3% of the robustness-measure range, noting that the range of the robustness measure is [0.5..8.0]. These results show that SOHOSim is an accurate simulator and a good proxy for a physical testbed.

Finding. The answer to RQ2 is that there is no statistically significant difference between SOHOSim and SOHOHW. The mean absolute error between the robustness measure values obtained from SOHOSim and SOHOHW is 3%.

5.4.3 Threats to Validity

Construct and external validity are the validity aspects most relevant to our evaluation.

5.4.3.1 Construct validity

The main consideration in relation to construct validity is the degree of accuracy of our simulator, SOHOSim. The risk of SOHOSim not being representative of the real world is mitigated by RQ2, where we show that SOHOSim behaves similarly to a physical testbed.

5.4.3.2 External validity

While our evaluation of ENRICH is based on a single case study, our experimental setup reflects a common SOHO setting, where the office connects to the Internet through a single traffic-shaping-enabled router. Strengthening external validity will require additional networking case studies using different testbeds and traffic mixes, to assess how consistently ENRICH identifies non-robust input regions across deployments. More broadly, extending ENRICH beyond networking would require defining a domain-specific notion of "small perturbations" over inputs. For Simulink-based control models, perturbations could be small bounded changes to continuous signals or reference setpoints (for example, slight adjustments to sensor readings, target altitude, or controller gains) over short time windows. For autonomous driving systems, perturbations could be small changes to the driving scene or scenario parameters (for example, mild shifts in lighting or weather, small variations in initial position and speed, or slight changes to nearby agents' trajectories) that preserve the scenario's intent while probing stability.

5.5 Lessons Learned

In this section, we reflect on two lessons learned from our collaboration with RabbitRun and the development of SOHOSim and ENRICH.

Lesson 1. Simulation as a way to discover unknown/undocumented behaviours. SOHOSim, in addition to enabling non-robustness analysis, helped our industry partner with the identification of unknown or undocumented behaviours. In particular, the multiple rounds of experimentation we conducted with RabbitRun using SOHOSim led to the following observations: (1) The total bandwidth allocated to CAKE should be configured such that it is not limiting the actual maximum network bandwidth. (2) The class priorities in CAKE are inversely related with the threshold ranges for the classes. (3) The quality of experience in each CAKE class depends not only on the priority of that class, but also on the bandwidth of the flows passing through the class. A key reason we could derive such high-level observations, while treating NTSS as an opaque box, is that SOHOSim is a *system-level* simulator. The main lesson learned here is that system-level simulators, in addition to fulfilling their analytical purpose (in our case, analysis of non-robustness), can be useful tools for exploration and identifying unknown/undocumented behaviours of complex systems.

Lesson 2. Non-robustness does not imply faultiness. For network systems, non-robustness is inevitable when bandwidth is constrained. As demand increases, the available bandwidth is eventually exceeded. No matter how well-designed an NTSS is, if overwhelmed, its quality of service eventually transitions to being robustly bad. This means that non-robustness is to be expected as one crosses the boundary between robustly good and robustly bad

regions. An important lesson in our study context is that non-robust regions of the input space are best *not* treated as faulty regions; that is, the existence of non-robust regions should not prompt fixes to the NTSS implementation. Instead, non-robust regions should be treated as situations that applications should attempt to steer clear of.

Taking the above lesson one level further, we believe that when non-robustness cannot be avoided through predefined static mappings, one needs more advanced safeguards, e.g., dynamic reconfiguration and self-adaptation at run-time, to preserve the quality of experience for as long as theoretically feasible. Existing NTSS are not yet equipped with such dynamic features. This presents interesting opportunities for applying ideas from self-adaptive systems to NTSS.

5.6 Summary

In this chapter, we proposed an approach that combines software testing and machine learning to generate input constraints that characterize a system's non-robust behaviours. We instantiated and empirically evaluated our approach over a novel case study from the network domain. This case study, which is concerned with a network traffic-shaping system, was conducted in collaboration with an industry partner, RabbitRun Technologies. Our approach accurately characterizes non-robust test inputs of NTSS by achieving a precision of 84% and a recall of 100%, significantly outperforming a standard baseline.

Chapter 6

Automated Test Validators for Flaky Cyber-Physical System Simulators

In this chapter, we present automated test validators designed to address the challenges of testing CPS with flaky and computationally expensive simulators. We introduce assertion-based test validators, which are sets of logical and arithmetic predicates over the inputs of the system under test. These validators learn conditions that capture (i) violations of requirement preconditions, (ii) inputs that lie outside the ODD, and (iii) nominal or low-risk scenarios in which the requirement is trivially satisfied. Whenever a test input satisfies one of these conditions, the validator can issue a pass or fail verdict directly from the input and the corresponding assertion, so simulator execution can be forgone. Only inputs that do not satisfy any learned condition are sent to the simulator, ensuring that expensive executions are reserved for tests that meaningfully exercise the system under test. This chapter details the formulation of these validators, the methods for generating them using

genetic programming and interpretable machine learning, and an extensive evaluation of their accuracy, robustness to flakiness, and alignment with reference documentation across the case studies in Chapter 3.

6.1 Introduction

As discussed in Section 1.2, some tests do not meaningfully exercise the SUT because they violate requirement preconditions, exceed ODD limits, or correspond to safe, low-risk scenarios. We refer to a mechanism that filters out such tests as a *test input validator*, or *test validator* for short.

An input filtered out by the validator may be well-formed and within the system's input domain, yet not useful for exercising the SUT's behaviour. This notion of validity aligns with recent work in deep learning (DL) testing [RT23, GAT+25, DDS21], where tests outside the DL model's training distribution are considered invalid. Even though an out-of-distribution test may be meaningful on its own, any failure resulting from it would likely be due to the DL model's unfamiliarity, not a fault.

Validity for DL systems has primarily been studied for image inputs [RT23, GAT+25, DDS21, SWCT20]. We extend this notion to CPS, which are typically tested using simulators that make testing expensive and time-consuming. Moreover, simulator non-determinism from environmental variability and stochastic processes can cause identical inputs to produce different outputs, leading to *flaky* test outcomes [BKB+23, NHG21, KPT24, LHEM14, PKHM21, SOW+20, DSM21]. For flaky tests – tests that pass or fail non-deterministically

– a single execution is often insufficient to determine the true outcome. The need for repeated executions and costly CPS simulations highlights the importance of a test validator that filters out non-essential test inputs for the SUT.

Tests that vacuously pass because they violate the SUT's preconditions, or trivially pass because they correspond to the SUT's nominal and low-risk conditions, can create a false sense of confidence in the system's trustworthiness. For example, vacuity may occur when testing an autopilot function in a scenario where the autopilot is never engaged, while a low-risk case may involve testing an autonomous driving system (ADS) collision requirement in a scenario where there are no nearby objects with which a collision is possible. Similarly, tests that fail because they violate the SUT's preconditions or exceed the ODD limits misrepresent trustworthiness by generating spurious errors. For example, to ensure the ascent requirement of an autopilot, a precondition is that sufficient initial throttle must be provided; otherwise, the test will fail. Likewise, an ADS travelling over 90 km/h at night in dense traffic exceeds its ODD limits and requires driver intervention.

Existing standards and specifications often describe preconditions and ODDs qualitatively but rarely define them with the precision needed to construct automated test validators. For example, a router specification may state that performance degrades under heavy streaming traffic without specifying which differentiated-services classes are affected or the threshold at which such degradation begins. To determine constraints and threshold bounds needed for constructing test validators, one must empirically infer input constraints that delineate nominal, boundary, and invalid regions of operation.

Prior data-driven approaches have inferred environmental assumptions and preconditions from simulation data using techniques such as genetic programming (GP) [Luk13] and interpretable machine learning (ML) models (e.g., decision trees) [KPAB22, GKH+20, KHSZ20, JCNS24, GMN+21, JNSS23]. However, these methods typically (1) treat inferred conditions as independent rules rather than as components of a test validator, and (2) overlook the impact of simulator flakiness on the reliability of the inferred rules. To our knowledge, no prior work has examined how label inconsistencies resulting from flaky test outcomes affect the robustness of learned input conditions.

Contributions. We propose a data-driven method for constructing test validators. Each validator is a set of assertions – arithmetic and logical predicates over system inputs – that explains the pass/fail outcomes observed in simulation-based testing. Assertions with high predictive confidence of pass or fail verdicts are likely to delineate input regions corresponding to nominal or low-risk conditions, violations of ODD boundaries, or unmet preconditions. We refer to our test validators as *assertion-based test validators* and infer them using (1) GP, which evolves assertions through fitness optimization, and (2) interpretable ML methods, namely decision trees (DT) and decision rules (DR), which extract human-readable conditions from training data. As described below, we ensure three key properties of these validators: (1) consistency of inferred verdicts, (2) effectiveness in learning interpretable assertions from training data, and (3) applicability to signal-based CPS:

(1) Combining assertions into a set requires ensuring that the set remains consistent in the verdicts it issues. Assertions from DT are consistent by construction, whereas those from DR and GP may be conflicting, since these techniques generate assertions for passing

and failing behaviours independently. *We propose a pruning mechanism to prevent test validators from assigning conflicting verdicts (Section 6.3.3).*

(2) *To improve the effectiveness of the test validators generated using GP, we use spectrum-based fault-localization (SBFL) ranking formulas as the fitness functions of GP (Section 6.3.1.2).* Specifically, we adopt three well-known SBFL ranking formulas from the literature [AZVG07, JH05, NLR11, LNB+16], namely *Ochiai*, *Tarantula*, and *Naish*. These formulas rank the suspiciousness of program statements based on their involvement in passing and failing executions. This ranking mechanism aligns with our goal of deriving assertions from training data that differentiate passing from failing behaviours.

(3) *To show the applicability of our test validators to signal-based CPS, we present a formal characterization of the expressive power of these validators in capturing common CPS signal properties (Section 6.4).* We formally show that, for CPS with piecewise-constant input signals, our assertions capture all logical operators as well as the "globally" temporal operator from Signal Temporal Logic (STL) [MN04]. This level of expressiveness is sufficient to capture 85 of the 98 requirements in the Lockheed Martin benchmark [loc25]. The assumption of piecewise-constant input signals is widely adopted in CPS benchmarks and case studies in the literature [loc25, cru25, clu25, gui25, dcm25, KFA+24].

Findings. Our evaluation focuses on two aspects: (1) the soundness and effectiveness of the proposed test validators in identifying violations of ODD limits, preconditions, and low-risk situations; and (2) their robustness to flakiness in training data obtained from simulators. To assess soundness and effectiveness, we measure both *accuracy* – how accurately the inferred test validators capture the input conditions that cause the SUT to pass or fail – and

alignment – the degree to which the learned assertions align with the descriptions of pre-condition violations, ODD limits, and low-risk scenarios provided in technical standards as well as in empirical and expert-validated studies [HJTM18,CAK25,ITU25,Pra21,YSYA19, CWX24, LSWH24, WAYZ25, WAWZ25, BNPR20, BES+23, ANBS18, aut25, Adm09]. We measure *robustness* as the variation in prediction accuracy across multiple test validators trained on datasets with identical test inputs but inconsistent pass or fail verdicts due to flakiness. Our robustness analysis aims to determine whether flaky tests – due to potentially unreliable verdicts – should be excluded from the initial training set used to develop test validators, or whether they can remain because their inclusion has minimal impact on the accuracy of the resulting test validator.

Our evaluation compares GP-based and ML-based approaches for constructing test validators with respect to accuracy, robustness, and alignment, across five case-study systems spanning networking, autonomous driving, and aerospace domains (Section 6.5). The main findings are summarized as follows:

Accuracy: Using GP with Ochiai is most effective for generating assertion-based test validators. Test validators generated by GP with Ochiai are significantly more accurate than those generated by GP with Tarantula or Naish, as well as those generated by DT and DR. Further, GP with Ochiai misclassifies fewer failing tests as passing compared to other techniques, thus reducing the likelihood of masking failures as passes. Overall, our test validators with confidence above 90% achieve an average accuracy of 89% across case studies in aerospace, networking, and autonomous driving.

Robustness: When using GP with Ochiai, the average variation in test validator accuracy is approximately 4%. This indicates that the impact on the prediction accuracy of test validators, caused by pass or fail label inconsistencies, is on average 4%. If such variation is acceptable, practitioners can forgo removing flaky tests – which require multiple test executions – from the initial training set, thus reducing costs.

Alignment: Based on a systematic human-subject study, we show that, on average, 80% of test-validator assertions with confidence above 90% inferred by GP align with descriptions of preconditions, nominal conditions, and ODD-limit violations in technical standards and empirical or expert-validated studies [HJTM18, CAK25, ITU25, Pra21, YSYA19, CWX24, LSWH24, WAYZ25, WAWZ25, BNPR20, BES+23, ANBS18, aut25, Adm09]. Another 8.7% partially match these sources, while the remaining 11.2% describe conditions not covered in the documentation but deemed plausible low-risk situations or ODD-violation scenarios by the human evaluators in our study. Finally, none of the assertions with confidence higher than 90% in our study contradicted the documentation.

Our full **replication package** is available online [git25].

Organization. Section 6.2 defines assertion-based test validators, presents their construction and provides an overview of their evaluation. Section 6.3 describes our data-driven framework for deriving assertion-based test validators, presents alternative approaches using GP and ML to generate assertion-based test validators, and introduces our pruning strategy for obtaining a consistent set of assertions as a test validator. Section 6.4 presents assertion-based test validators for signal-based CPS, focusing on specifying assertions over signals and their expressiveness, with a running example. Section 6.5 presents

an evaluation of the alternative approaches (GP and ML) for generating assertion-based test validators. Section 6.6 summarizes this chapter.

6.2 Assertion-based Test validators

We define and illustrate assertion-based test validators in Section 6.2.1, and then provide an overview of their construction and evaluation in Sections 6.2.2 and 6.2.3, respectively.

6.2.1 Defining Test Validators

An *assertion* for a system S, is an expression of the form *condition* $\Rightarrow$ *verdict*, where *condition* is a logical predicate over the inputs of S and *verdict* is either pass or fail. For a test input t, the assertion *condition* $\Rightarrow$ *fail* implies that if t satisfies *condition*, then the verdict for t is fail. Similarly, the assertion *condition* $\Rightarrow$ *pass* implies that if t satisfies *condition*, then the verdict for t is pass. We refer to an assertion with a fail verdict as a *fail assertion* and an assertion with a pass verdict as a *pass assertion*.

Figure 6.1 shows two passing assertions (a_1 and a_3) and one failing assertion (a_2) for a simplified ADS. These assertions are defined over three ADS inputs: ego-vehicle speed (*speed*), time of day (*time_of_day*), and traffic density (*traffic_density*). For instance, the pass assertion a_1 specifies that when the ego vehicle is driving at $\leq$ 20 km/h and there are no nearby vehicles, the scenario is low risk and the ADS is very unlikely to violate the collision requirement. Similarly, the fail assertion a_2 indicates that when the ego vehicle is driving faster than 90 km/h at night under heavy traffic, the scenario is high risk, with

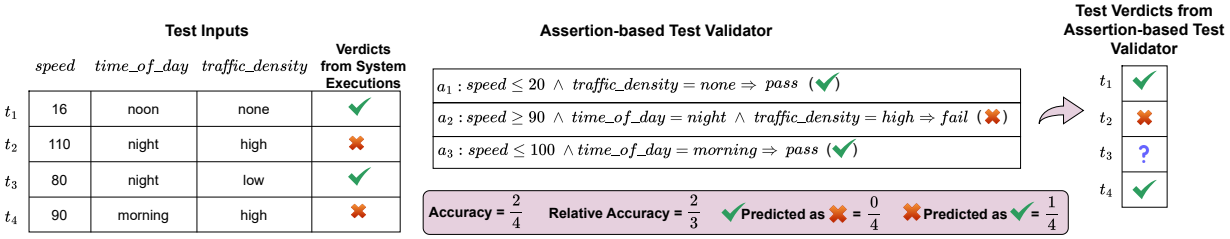


Figure 6.1: An assertion-based test validator for a simplified ADS with inputs *speed*, *time_of_day* and *traffic_density*, along with an example set of test inputs for this system. In this figure, ✓ indicates the pass verdict, ✗ indicates the fail verdict and **?** indicates that the test validator is inconclusive.

a high likelihood of collision, because the situation is likely beyond the control of the underlying ADS.

Definition 6.1 (**Test validator**). *A* test validator $\mathcal{V}$ *for a system* S *is a set of assertions defined for* S *whose members are pairwise* consistent. *Two assertions* $cnd \Rightarrow v$ *and* $cnd' \Rightarrow v'$ *are* consistent *if* $v \neq v'$ *implies that the conjunction* $cnd \wedge cnd'$ *is unsatisfiable (UNSAT).*

Definition 6.1 ensures that a test validator is consistent in the verdicts it produces. While a test input t may satisfy the conditions of multiple assertions, all such assertions must agree on the verdict – either all pass or all fail. For example, the assertions a_1, a_2 and a_3 in Figure 6.1 constitute a test validator as they are pairwise consistent. However, assertion a_4 : $speed > 15 \wedge time\_of\_day = morning \Rightarrow fail$, not shown in the figure, is inconsistent with a_3, since the conjunction of the conditions of a_3 and a_4 is satisfiable, and they predict opposing verdicts (pass and fail, respectively).

The verdict assigned by an assertion-based test validator $\mathcal{V}$ for a test input t is determined as follows:

$$\begin{cases} \mathit{fail} & \text{if } \exists(\mathit{cnd} \Rightarrow \mathit{fail}) \in \mathcal{V} \ s.t. \ t \models \mathit{cnd} \\ \mathit{pass} & \text{if } \exists(\mathit{cnd} \Rightarrow \mathit{pass}) \in \mathcal{V} \ s.t. \ t \models \mathit{cnd} \\ \mathit{inconclusive} & \text{otherwise} \end{cases}$$

Specifically, if t satisfies the condition of *any* pass assertion $(\mathit{cnd} \Rightarrow \mathit{pass})$ in $\mathcal{V}$, then the verdict for t is *pass*. Conversely, if t satisfies the condition of *any* fail assertion $(\mathit{cnd} \Rightarrow \mathit{fail})$ in $\mathcal{V}$, then the verdict for t is *fail*. If t does not satisfy the condition of any assertion in $\mathcal{V}$, the verdict is considered *inconclusive*. In other words, inconclusive tests are those that the validator cannot classify as passing or failing. These tests are suitable candidates for execution on the SUT, as they are likely to exercise its behaviour. For example, Figure 6.1 shows four test inputs t_1 to t_4 for the ADS. The test validator in the figure implies that t_1 and t_4 pass (satisfying a_1 and a_3), t_2 fails (satisfying a_2), and t_3 is inconclusive.

6.2.2 Test-validator Construction and Verdict Thresholds

To construct assertion-based test validators, we adopt a data-driven approach that infers assertions from training data consisting of test inputs and their corresponding pass/fail verdicts. Because these assertions are learned from finite datasets, each is associated with a confidence level – expressed as a probability – that reflects the assertion's reliability in predicting a verdict based on the training data. Specifically, the confidence level of an

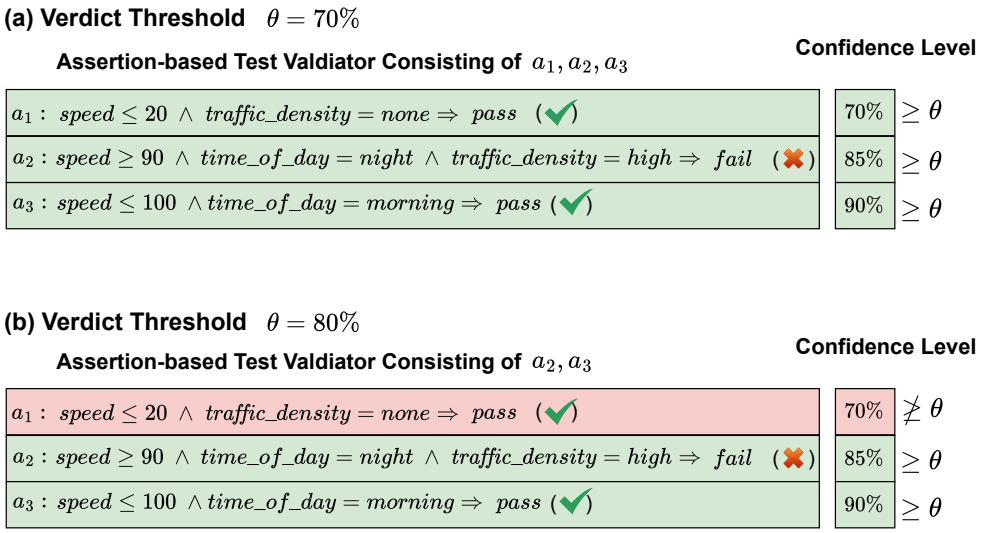


Figure 6.2: Confidence levels for assertions a_1, a_2 and a_3 in Figure 6.1, where (a) at a verdict threshold of $\theta = 70\%$, all assertions are included in the test validator, and (b) at a verdict threshold of $\theta = 80\%$, only assertions a_2 and a_3 are included in the test validator.

assertion is defined as its precision in classifying pass or fail tests in the training data. Given an assertion and its associated confidence, we must decide whether the confidence level is sufficient for the assertion to be included in a test validator for verdict prediction. To this end, we introduce a user-defined lower bound, called the *verdict threshold*:

Definition 6.2 (**Verdict threshold**). *Let $\mathcal{V}$ be an assertion-based test validator constructed using a data-driven approach. The verdict threshold θ is a minimum confidence level such that only assertions whose confidence levels meet or exceed θ are retained in $\mathcal{V}$ for predicting test verdicts.*

For example, Figure 6.2 shows two test validators, each consisting of the assertions a_1, a_2, and a_3 with confidence levels of 70%, 85%, and 90%, respectively. Assuming that the verdict threshold θ is set to 70% for the test validator in Figure 6.2(a), all assertions a_1

through a_3 are included. However, in Figure 6.2(b), when the verdict threshold θ is raised to 80%, only assertions a_2 and a_3 are included.

6.2.3 Metrics for Test Validators

As discussed in Section 6.1, we evaluate test validators in terms of their accuracy, robustness, and alignment. Alignment is assessed *qualitatively* by comparing test validators with system specifications, technical standards and empirical results in the literature. We assess accuracy and robustness *quantitatively* against test sets obtained through system execution using the following metrics:

We measure prediction accuracy in two ways: (1) using the *accuracy* metric, which measures the percentage of correctly predicted verdicts across all tests (capturing losses from both incorrect and inconclusive predictions), and (2) using the *relative accuracy* metric, which measures the percentage of correctly predicted verdicts restricted to conclusive predictions (capturing only incorrect predictions). In addition, we report the *misprediction rate* metric, defined as the percentage of incorrectly predicted verdicts across all tests, distinguishing between (1) passing tests wrongly predicted as failing and (2) failing tests wrongly predicted as passing.

For example, based on the actual (ground-truth) test verdicts shown in Figure 6.1, the predicted verdicts for t_1 and t_2 are correct, the verdict for t_4 is incorrect, and the verdict for t_3 cannot be predicted conclusively. Thus, the accuracy of the test validator in Figure 6.1 is $\frac{2}{4}$ and its relative accuracy is $\frac{2}{3}$. In addition, the rate of the pass tests wrongly predicted as fail and the rate of fail tests wrongly predicted as pass are $\frac{0}{4}$ and $\frac{1}{4}$, respectively.

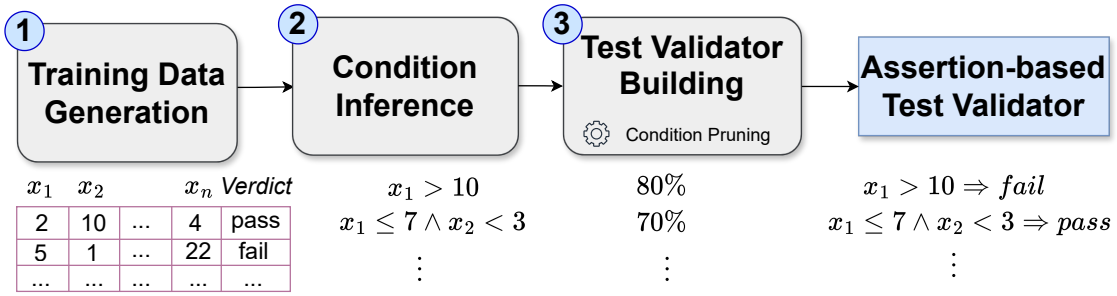


Figure 6.3: Our approach for deriving assertion-based test validators, GenTV.

For robustness, we measure the variation in prediction accuracy across multiple test validators trained on datasets with identical test inputs but inconsistent pass/fail verdicts due to flakiness. A robust test validator should maintain consistent accuracy, showing minimal variation despite such inconsistencies in the training data.

In Section 6.5, we evaluate different techniques for generating assertion-based test validators using the metrics introduced in this section, with RQ2 focusing on accuracy, RQ3 focusing on robustness, and RQ4 focusing on alignment.

6.3 Generating Assertion-based Test validators

Figure 6.3 shows an overview of GenTV, our framework for generating assertion-based test validators. The framework has three steps. The first step of GenTV uses adaptive random testing [Luk13] to generate a training set of test inputs. To label these test inputs as either pass or fail, they are executed on the SUT.

The second step of GenTV uses the training data generated in the first step to learn conditions for assertions. Recall from Section 6.2.1 that conditions are logical predicates

defined over SUT's inputs. For example, $speed \leq 20 \wedge traffic\_denisty = none$ is a condition for assertion a_1 in Figure 6.1. This step identifies conditions that best explain pass elements or fail elements in the training data. We consider two alternatives for condition inference: (1) using genetic programming (GP) presented in Sections 6.3.1, and (2) using Decision Trees (DT) and Decision Rules (DR) presented in Section 6.3.2. The third and final step of GenTV, described in Section 6.3.3, filters out assertions whose confidence levels fall below a specified verdict threshold θ, and then applies a pruning algorithm to ensure that the remaining assertions in the test validator are mutually consistent.

6.3.1 Condition Inference by Genetic Programming (GP)

The first alternative we consider to generate conditions defined over the SUT's input variables is GP. GP requires a grammar to define its candidate solutions, which in our work, are conditions based on the SUT's input variables. Below, we first present the grammar that GenTV uses for GP, and then explain how GP learns assertion conditions based on the training set generated in the first step.

6.3.1.1 Grammar Specification

Since our work targets CPS, where inputs are typically numeric [JCNS24, GMN+21], we adopt the grammar in Figure 6.4 (denoted $\mathcal{G}$), previously used to specify environmental assumptions in CPS [GMN+21] and control logic in network systems [LNS24]. Grammar $\mathcal{G}$ generates conditions that are either conjunctions of relational expressions over arithmetic terms or disjunctions of such conjunctions.

$$
\begin{aligned}
\text{or-term} &::= \text{or-term} \vee \text{or-term} \mid \text{and-term} \\
\text{and-term} &::= \text{and-term} \wedge \text{and-term} \mid \text{rel-term} \\
\text{rel-term} &::= \text{exp} < 0 \mid \text{exp} \leq 0 \mid \text{exp} > 0 \mid \text{exp} \geq 0 \mid \text{exp} = 0 \mid \text{exp} \neq 0 \\
\text{exp} &::= \text{exp} + \text{exp} \mid \text{exp} - \text{exp} \mid \text{exp} * \text{exp} \mid \text{exp} \,/\, \text{exp} \mid \text{const} \mid \text{cp}
\end{aligned}
$$

Figure 6.4: Syntactic rules of the grammar (denoted by $\mathcal{G}$) that define the assertions over system input variables.

The symbol | in the above grammar separates alternatives, const is an ephemeral random constant generator [Vee13] and cp represents an input variable of the SUT. This grammar generates conditions that are either conjunctions of relational expressions over arithmetic terms or disjunctions of such conjunctions.

6.3.1.2 Condition Learning

We use the standard GP workflow explained in Section 2.2.2 and shown in Algorithm 2.1 to infer conditions that explain pass and fail verdicts in the training set TS, generated in the first step of GenTV (Figure 6.3). The fitness function of GP evaluates how well each candidate condition explains the pass and fail verdicts in TS. Specifically, GP employs two complementary fitness functions: one aimed at generating conditions that explain the fail results, and another aimed at generating conditions that explain the pass results. We run GP separately with each fitness function to independently derive the pass and fail conditions.

Following the standard practice for expressing metaheuristic search problems [HMdSY12], we define the individual representation, the genetic operators, and the fitness function of GP:

Individual Representation. A GP individual represents a condition created by following the grammar $\mathcal{G}$ in Figure 6.4. The initial population (P_0) is formed by randomly constructing parse trees employing the grow method [PLM08].

Genetic Operators. We use one-point crossover as well as one-point mutation for population breeding. These operators are adopted from the prior application of GP to similar applications, particularly in learning environmental assumptions for CPS [GMN+21]. To ensure that the generated candidate solutions comply with GP's grammar, we verify them during breeding and mutation and discard any invalid ones.

Fitness Functions. We use three spectrum-based fault localization (SBFL) formulas, Tarantula [JH05], Ochiai [AZVG07], and Naish [NLR11], as GP fitness functions. Section 2.3 presents these formulas for the fail verdict, noting that the fitness functions for the pass verdict are the duals of those for the fail verdict.

To use the SBFL functions for condition inference, we interpret c as a candidate condition within the GP's population. The functions $c_p(c)$ and $c_f(c)$ then compute, respectively, the number of passing and failing test inputs in the training set TS that satisfy c. Figure 6.5 shows how c_p and c_f are calculated for two GP individuals, namely c_1 and c_2. As shown in the figure, t_1 satisfies c_1, and t_2 and t_4 satisfy c_2. Hence, we have $c_p(c_1) = 0$, $c_f(c_1) = 1$, $c_p(c_2) = 1$ and $c_f(c_2) = 1$. Any of the SBFL ranking functions in Section 2.3 can then be used to compute a fitness value for conditions c_1 and c_2.

SBFL functions assign high values to conditions met by many failing and few passing test inputs in the training set. By using SBFL ranking functions as fitness functions, our GP-based approach selects conditions that are more likely to explain and characterize

(a) Training Set (TS)

	Road Angle	Max Speed	Outcome
t_1	20	95	Fail
t_2	65	10	Fail
t_3	15	80	Pass
t_4	70	5	Pass

(b) Fitness Values

GP Individual	t_1	t_2	t_3	t_4	c_p	c_f
c_1: *RoadAngle* < 40 ∧ *MaxSpeed* > 90	✓				0	1
c_2: *RoadAngle* > 60		✓		✓	1	1

Figure 6.5: Computing $c_p(c)$ and $c_f(c)$ in SBFL fitness functions in Section 2.3 for our GP-based condition inference.

failures effectively. For fitness functions that explain passing test cases, we swap c_f with c_p and swap TS_p with TS_f in the functions in Section 2.3.

6.3.2 Condition Inference by Interpretable ML

The second alternative we consider for generating conditions defined over the SUT's input variables is interpretable ML. Specifically, we consider decision trees (DT) and decision rules (DR) as two forms of interpretable supervised ML models. Other interpretable supervised ML models such as linear, logistic, and polynomial regression assume linear or polynomial relations between the input variables and the test verdict [Mol20]. Hence, these models are limited in their ability to capture the conditions required by our test validators, which take the form of non-linear and rule-based constraints between input variables and test verdicts.

Inferring conditions using either DT or DR involves two steps: (1) *Feature Engineering:* We define input features for learning, initially using the system's input variables as default features. However, if only these variables are used, DT and DR can learn only simple

conditions that relate a single variable to a constant through a relational operator. For systems where the relationship between input variables and test outcomes is more complex, feature engineering becomes crucial. This process involves creating features that combine input variables using mathematical operators, allowing DT and DR to learn conditions based on these arithmetic combinations, similar to those generated by the grammar $\mathcal{G}$ in Figure 6.4. (2) *Model Training and Condition Generation:* With the input features established, we train the DT or DR models using the training set *TS* generated in the first step of GenTV. DT and DR generate conditions as disjunctions of conjunctions of expressions that relate input features to constants, similar to conditions generated by the grammar $\mathcal{G}$. These models can produce conditions for both pass and fail verdicts.

6.3.3 Test Validator Building

In the third and final step, we construct consistent assertion-based test validators from the conditions generated in the second step of GenTV. To do so, we first calculate each condition's confidence level based on its precision in classifying pass or fail tests in the training set. Specifically, if condition *cnd* is associated with a fail (or pass) verdict, its confidence level is the percentage of actual fail (or pass) test inputs in the training set that satisfy *cnd*, relative to all test inputs that satisfy *cnd*. For instance, if the condition c_2 in Figure 6.5 is associated with a pass verdict, its confidence level is 50% because among the two tests in the training set that satisfy c_2 (i.e., t_2 and t_4), only t_4 has a pass verdict. Then, we retain only those conditions whose confidence levels meet or exceed the user-defined

verdict threshold θ. Recall from Definition 6.2 that θ specifies the minimum confidence level required for a condition to be included in the assertion-based test validator.

Next, we apply a pruning strategy to obtain a consistent set of assertions in the test validator. Recall from Definition 6.1 that a consistent test validator ensures no conflicting assertions exist, where one assertion indicates a test input passes while another indicates it fails. While DT-inferred assertions are consistent by construction, those inferred by GP or DR may require pruning to maintain consistency. Below, we first establish the necessary notation and definitions, and then describe our pruning strategy for obtaining a consistent set of assertions.

Definition 6.3 (**Bipartite Graph based on Assertion Conditions**). *Let A be a set of assertions. We define the bipartite graph $\mathcal{B} = (V, E)$ corresponding to A as follows: each assertion in A corresponds uniquely to a vertex in V. The set V is partitioned into two disjoint subsets V_p and V_f, where V_p represents conditions of the pass assertions in A, and V_f represents conditions of the fail assertion in A. An edge $(x, y) \in E$ connects a vertex $x \in V_p$ to a vertex $y \in V_f$ if and only if the conjunction of their associated assertion conditions is satisfiable (SAT). More precisely, an edge between x and y indicates that the pair of assertions corresponding to x and y together represent an inconsistency within A since they, respectively, represent a passing assertion and a failing assertion that can simultaneously hold for some test input.*

Figure 6.6 shows a bipartite graph where a_1, a_2, a_3, and a_4 represent conditions of pass assertions (i.e., they belong to V_p), and a'_1, a'_2, and a'_3 represent conditions of fail assertions (i.e., they belong to V_f). The edges in this graph represent pairs of pass and

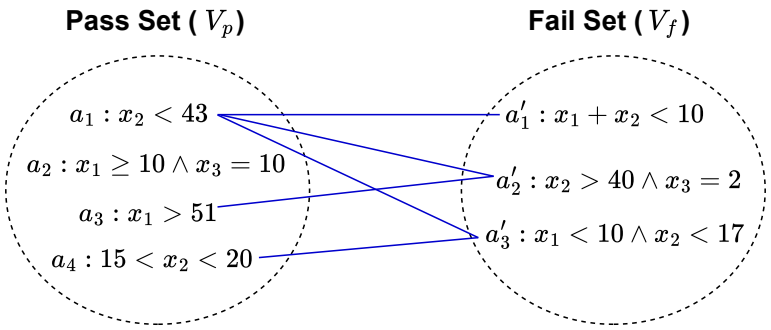


Figure 6.6: Pruning inconsistent assertions from test validators using a bipartite graph representation. In this figure, V_p denotes pass-class conditions, while V_f denotes fail-class conditions.

fail assertions whose conjunctions are satisfiable. For example, there is an edge between a_1 and a'_1 because $x_2 < 43 \wedge x_1 + x_2 < 10$ is SAT.

Let $x \in V$ be a vertex. We denote the *length* of the condition associated with x by $len(x)$, defined as the number of arithmetic and logical operators present in that condition. For example, in Figure 6.6, $len(a_1) = 1$ and $len(a_2) = 3$. We also denote the *degree* of a vertex x by $deg(x)$, defined as the number of edges connected to x. For instance, in Figure 6.6, $deg(a_1)$ is three, indicating that the passing assertion corresponding to a_1 is inconsistent with the failing assertions corresponding to a'_1, a'_2, and a'_3.

We now present our pruning method, shown in Algorithm 6.1, which aims to eliminate inconsistencies (i.e., edges) between pass and fail conditions by removing vertices (i.e., conditions) from V_p or V_f. Since there are multiple ways to eliminate inconsistencies, resulting in alternative consistent sets of assertions, we devise heuristics in Algorithm 6.1 whose goal is to obtain a consistent set of assertions while minimizing the number of removed vertices. Algorithm 6.1 takes a potentially inconsistent set of assertions A and

Algorithm 6.1 Pruning strategy to obtain a consistent set of assertions

Input A: A (potentially inconsistent) set of assertions
Output A': A consistent subset of A

1: Let $\mathcal{B} = (V, E)$ be a bipartite graph created based on Definition 6.3.
2: **while** $E \neq \emptyset$ **do**
3: $S \leftarrow \{x \in V \mid deg(x) \geq 1 \wedge \forall y \in V : len(x) \leq len(y)\}$
4: **if** $\mid S \mid == 1$
5: $vertexToRemove \leftarrow$ select the vertex in S
6: **else**
7: $H \leftarrow \{x \in S \mid \forall y \in S : deg(x) \geq deg(y)\}$
8: **if** $\mid H \mid == 1$
9: $vertextToRemove \leftarrow$ select the vertex in H
10: **elseif** $H \cap V_p \neq \emptyset$
11: $vertexToRemove \leftarrow$ randomly select a vertex in $H \cap V_p$
12: **else**
13: $vertexToRemove \leftarrow$ randomly select a vertex in H
14: **end**
15: **end**
16: $V \leftarrow V \setminus \{vertexToRemove\}$
17: $E \leftarrow E \setminus \{(u, v) \in E \mid u = vertexToRemove \vee v = vertexToRemove\}$
18: **end**
19: $A' \leftarrow \{x \Rightarrow fail \mid x \in V_f\} \cup \{x \Rightarrow pass \mid x \in V_p\}$
20: **return** A'

generates a consistent subset A' of assertions from A. Based on Definition 6.3, the algorithm represents A as a bipartite graph $\mathcal{B} = (V, E)$, where each assertion condition becomes a vertex in V and each satisfiable pass-fail pair is an edge in E (line 1 in Algorithm 6.1). We use the Z3 SMT solver [DMB08] to check the satisfiability of the conjunctions of all pairs of pass and fail conditions to establish edges.

In the while-loop from lines 2 to 18, Algorithm 6.1 iteratively removes vertices until there are no remaining edges between the pass (V_p) and fail (V_f) partitions. The loop begins by identifying the set S of vertices with a degree of at least one and the shortest length among such vertices (line 3 in Algorithm 6.1). If S contains exactly one vertex, that

vertex is selected and stored in the variable *vertexToRemove* (lines 4–5 in Algorithm 6.1) for removal at the end of the while-loop iteration. This is because the condition corresponding to this vertex is the least constrained, thus having a higher likelihood of conflicting with other conditions, making it a priority candidate for removal.

If multiple vertices exist in S, the algorithm computes the subset $H \subseteq S$ containing vertices with the highest degree (line 7 in Algorithm 6.1). If H contains exactly one vertex, this vertex is selected for removal (lines 8–9 in Algorithm 6.1). Otherwise, when multiple vertices exist in H, the algorithm prioritizes vertices belonging to the pass set (V_p) by randomly selecting one vertex in $H \cap V_p$ (lines 10–11 in Algorithm 6.1). This prioritization is motivated by the observation that the fail partition (V_f) typically contains fewer vertices, making fail vertices more valuable to retain. If no vertex in H belongs to the pass set, the algorithm randomly selects a vertex from H (line 13 in Algorithm 6.1). Having stored a vertex in *vertexToRemove*, this vertex and all its incident edges are removed from the graph (lines 16–17 in Algorithm 6.1). The assertions corresponding to the remaining vertices in V_p and V_f are collected into the set A', which is then returned (lines 19–20 in Algorithm 6.1).

Algorithm 6.1 ensures that the returned set A' is consistent. This is because at each iteration, the algorithm removes exactly one vertex with at least one incident edge. Since the graph $\mathcal{B} = (V, E)$ has a finite number of edges, the algorithm terminates after at most $|E|$ iterations. Upon termination, all edges – representing inconsistent pairs of assertion conditions – have been removed. Hence, the set A' is consistent.

For example, in Figure 6.6, vertex a_1 is removed first because it has the shortest condition and the highest degree. Next, a_3 is removed since it has the highest degree and

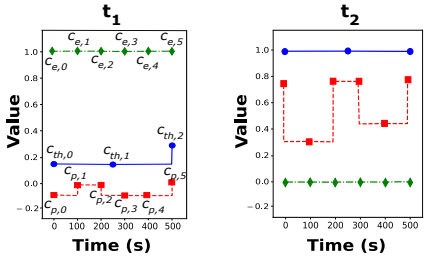
a condition shorter than those of the vertices with the same degree, i.e., a'_2, a_4 and a'_3. Finally, a_4 is removed as it belongs to the pass class and has the same length and degree as a'_3. After removing a_1, a_3 and a_4, the remaining conditions are consistent.

6.4 Test Validators for Signal-based CPS

In this section, we adapt Definition 6.1 to signal-based CPS. While Definition 6.1 defines assertion-based test validators for discrete-input CPS, signal-based systems require a formulation that accounts for inputs represented as signals. In the following, we introduce a running example to illustrate assertion-based test validators for such systems, explain how assertions are specified over signals, and discuss the expressiveness of these assertions in capturing signal properties.

6.4.1 Motivating Example

To illustrate assertion-based test validators for CPS with signal-based inputs, we use a simplified autopilot controller, AUTOPILOT, as our running example. AUTOPILOT has time-varying input signals: *throttle* (engine power adjustment), *pitchwheel* (nose tilt), and *apeng* (autopilot engagement state). When *apeng* indicates that the autopilot is engaged, AUTOPILOT uses the throttle and pitchwheel signals to issue actuator commands that control the aircraft's orientation and motion. AUTOPILOT has a requirement stating that when autopilot is engaged, the aircraft should reach a specified altitude within 500 seconds.



(a) Test Inputs

$\mathbf{a_1}$: $(\forall t \in [0, 300) : p(t) < 0.1) \wedge (\forall t \in [0, 500) : th(t) \leq 0.3 \wedge e(t) = 1.0) \Rightarrow \mathit{fail}$

$\mathbf{a_2}$: $(\forall t \in [200, 500) : p(t) < -0.5) \wedge (\forall t \in [0, 500) : e(t) = 1.0) \Rightarrow \mathit{fail}$

$\mathbf{a_3}$: $(\forall t \in [200, 500) : 0.2 < p(t) < 0.8) \wedge (\forall t \in [0, 500) : e(t) = 0.0) \Rightarrow \mathit{pass}$

(b) Assertion-based Test Validator

Figure 6.7: (a) Two test inputs, t_1 and t_2 for an autopilot system, with input signals *Throttle*, *PitchWheel*, and *APEng*; (b) An assertion-based test validator for this system. According to the test validator, t_1 fails and t_2 passes.

Figure 6.7 shows three assertions over AUTOPILOT's inputs: throttle (th), pitchwheel (p), and apeng (e). Note that these three signal variables (th, p, and e) are time dependent. Assertion a_1 characterizes a situation in which the autopilot is engaged but engine power is insufficient, violating the ascent requirement's precondition of sufficient thrust. Assertion a_2 characterizes a situation in which the autopilot is engaged while the aircraft's nose remains sharply pitched downward for 300 seconds, violating the precondition that the nose should point upward during ascent. Finally, assertion a_3 characterizes a safe situation in which the autopilot is disengaged, so the ascent requirement is vacuously satisfied.

Together, the assertions in Figure 6.7 form an assertion-based test validator for AUTOPILOT. Figure 6.7 further presents two test inputs (t_1 and t_2), each displaying signals for throttle, pitchwheel, and apeng. The assertions determine that t_1 fails while t_2 passes without executing the system.

Note that these three signal variables, p, $apeng$, and th, are defined over time, and the assertions in Figure 6.7 constrain their values over time. Below, we describe how the assertions over the signal variables in Figure 6.7 are generated by grammar $\mathcal{G}$ in Figure 6.4.

6.4.2 Assertions over Signals

Recall from Section 3.1 that a common approach for representing signals is by encoding them as sequences of equally spaced control points, where each index denotes a fixed time interval and each corresponding value approximates the signal's value during that time interval. Provided with the control points, the actual signals are constructed through interpolation. For example, in the test inputs in Figure 6.7, pitchwheel (p) and apeng (e) are represented using six control points (denoted by $c_{p,0}, \ldots, c_{p,5}$ for the pitchwheel and $c_{e,0}, \ldots, c_{e,5}$ for the apeng), while throttle (th) is represented using three control points ($c_{th,0}, \ldots, c_{th,2}$).

Assertion conditions over signals are generated according to the syntactic rules of grammar $\mathcal{G}$ in Figure 6.4 by using cp to represent signal control points. Furthermore, to ensure the well-formedness of the conditions, we constrain each arithmetic expression, exp, to contain only signal control points at the same position. For example, the condition $(c_{e,0} - c_{p,0} \geq 0) \wedge (c_{e,1} + c_{th,1} < 1)$ can be generated by our grammar and satisfies the constraint that each arithmetic expression must involve control points associated with the same position; $c_{e,0}$ and $c_{p,0}$ in the first expression are control points at position 0, and $c_{e,1}$ and $c_{th,1}$ in the second expression are control points at position 1.

6.4.3 Expressive Power of Assertions over Signals

We now discuss the expressive power of the assertions generated using grammar $\mathcal{G}$ in Figure 6.4 over signal control points. To do so, we first present a translation of assertion conditions defined over control points into constraints defined over signal variables directly.

Our translation consists of two sets of rewriting rules: The first set of rewriting rules, based on Menghi et al. [MNGB19], introduces a $\forall$ quantifier over each arithmetic expression, $\texttt{exp}$, while replacing signal control points with signal variables in these expressions. The second set of rules consists of standard logic rewriting rules: one for merging universal quantifiers over disjoint domains, and another for conjunction of universal quantifiers over the same domain.

Rules for replacing signal control points with signal variables [MNGB19]. Let $u : \mathbb{T} \to \mathbb{R}$ be an input signal. Suppose we represent u using the following n_u control points: $c_{u,0}$, $c_{u,1}$, $\ldots$, c_{u,n_u-2}, c_{u,n_u-1} such that each control point $c_{u,i}$ is positioned at position $i \cdot I$ where $I = \frac{b}{n_u-1}$. Let $\texttt{exp}$ be an arithmetic expression generated by the grammar $\mathcal{G}$ in Figure 6.4 in Section 6.3. Based on the definition of assertions over signals given above, $\texttt{exp}$ contains only signal control points in the same position j. This expression can then be rewritten into the following equivalent logical expression $\forall t \in [j \cdot I, (j+1) \cdot I) : \texttt{exp}'$ where $\texttt{exp}'$ is obtained by substituting each control point $c_{u,j}$ with the expression $u(t)$ representing the input signal u at time t. Note that this rewriting rule is valid because we assume that the interpolation function connecting the control points is piecewise constant.

For example, in Figure 6.7 in Section 6.4, consider the control points $c_{e,0}$ to $c_{e,5}$ for the apeng signal e, and the control points $c_{p,0}$ to $c_{p,5}$ for the pitchwheel signal p. The first control point of both signals is at time 0s, the second at 100s, the third at 200s, and so on. Now, consider the following condition over these control points which can be generated by our grammar:

$$(c_{e,0} - c_{p,0} \leq 20) \wedge (c_{e,1} + c_{p,1} < 0)$$

The above condition is rewritten into the following logical formula over apeng (e) and pitchwheel (p) signals based on the rule discussed above:

$$(\forall t \in [0, 100) : e(t) - p(t) \leq 20) \ \wedge \ (\forall t \in [100, 200) : e(t) + p(t) < 0)$$

Note that for the control points at position zero, i.e., $c_{e,0}$ and $c_{p,0}$, we quantify the variable t over the domain $[0, 100)$, and for the control points at position one, i.e., $c_{e,1}$ and $c_{p,1}$, we quantify the variable t over the next time slot $[100, 200)$.

Quantifier conjunction rules. After introducing universal quantifiers and replacing control points with signal variables, we apply the following standard logic rewriting rules:

$$(\forall t \in A : \texttt{exp}) \wedge (\forall t \in B : \texttt{exp}) \ \equiv \ \forall t \in A \cup B : \texttt{exp}$$
$$(\forall t \in A : \texttt{exp}) \wedge (\forall t \in A : \texttt{exp}') \ \equiv \ \forall t \in A : \texttt{exp} \wedge \texttt{exp}'$$

where $\texttt{exp}$ and $\texttt{exp}'$ are arithmetic expressions containing signals over the time domain $\mathbb{T} = [0, b]$, and A and B are two time domains that are subsets of $\mathbb{T}$.

Figure 6.8 illustrates the condition assertions generated using our grammar over the control points of the apeng (e), pitchwheel (p), and throttle (th) signals, along with the step-by-step translation of these conditions into logical formulas over their corresponding signal variables. For example, in Figure 6.8(b), the condition

$$c_{p,2} < -0.5 \wedge c_{p,3} < -0.5 \wedge c_{p,4} < -0.5 \wedge c_{e,0} = 1.0 \wedge c_{e,1} = 1.0 \wedge c_{e,2} = 1.0 \wedge c_{e,3} = 1.0 \wedge c_{e,4} = 1.0$$

is converted into the logical formula:

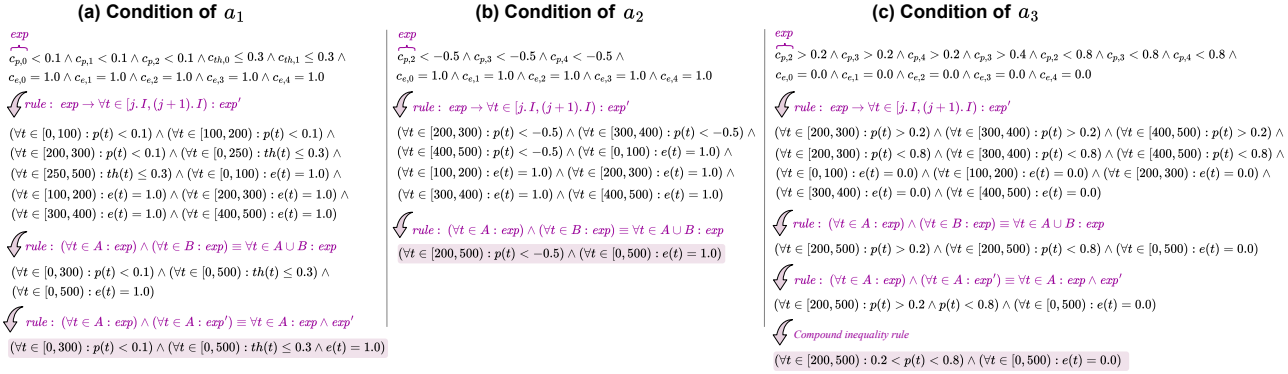


Figure 6.8: Step-by-step illustration of using rules to derive logical assertion conditions over the signals in Figure 6.7 in Section 6.4 from assertion conditions based on the control points of apeng (e), pitchwheel (p) and throttle (th).

$$(\forall t \in [200, 500) : p(t) < -0.5) \wedge (\forall t \in [0, 500) : e(t) = 1.0).$$

To perform this translation, we first rewrite each relational term (e.g., $c_{p,2} < -0.5$) in the condition into its logical equivalent using the rule $exp \rightarrow \forall t \in [j.I, (j+1).I] : exp'$ where j is the position of the control point and I is the sampling interval. We then apply the quantifier conjunction rules to combine the resulting quantified expressions into a single statement that covers the union of their time ranges.

Having provided the rewriting rules above, we can now discuss the expressiveness of the conditions obtained after applying the rewriting rules. Let $\bar{u} = u_1, \ldots, u_m$ be signals over $\mathbb{T}$, and let $T = \{t_1, \ldots, t_d\}$ be a set of time variables. Suppose we generate conditions over the control points of the signals in $\bar{u}$ using grammar $\mathcal{G}$ in Figure 6.4. After applying the two rewriting rule sets above, the generated formulas are expressible within the following logic fragment, denoted by $\mathcal{L}$:

$$\psi ::= \psi_1 \vee \psi_2 \mid \phi$$

$$\phi ::= \rho \sim 0 \mid \phi_1 \wedge \phi_2 \mid \forall t \in \langle n_1, n_2 \rangle : \phi$$

$$\rho ::= u(t) \mid r \mid \rho_1 + \rho_2 \mid \rho_1 - \rho_2 \mid \rho_1 \times \rho_2 \mid \rho_1 / \rho_2$$

where n_1, n_2 are non-negative real numbers including zero, $r \in \mathbb{R}$, $t \in \mathbb{T}$, $u \in \bar{u}$, $\sim$ is a relational operator in $\{<, \leq, >, \geq, =, \neq\}$, and $\langle n_1, n_2 \rangle$ is a *time interval* of $\mathbb{T}$ (i.e., $\langle n_1, n_2 \rangle \subseteq \mathbb{T}$). The symbols $\langle$ and $\rangle$ are equal to [or (, respectively to] or), depending on whether n_1, respectively n_2, are included or excluded from the interval.

We argue that any condition obtained by our grammar $\mathcal{G}$ and modified through our rewriting rules is a formula in $\mathcal{L}$. Conversely, any formula $\varphi \in \mathcal{L}$ that satisfies the following two conditions corresponds to a condition that can be generated by the grammar $\mathcal{G}$:
(1) φ is closed, i.e., does not contain any free occurrence of the variable t, and (2) φ does not involve any nested use of the $\forall$ quantifier.

The argument follows by structural induction and noting that the nonterminal ψ in $\mathcal{L}$ corresponds to or-term in $\mathcal{G}$; the nonterminal ϕ in $\mathcal{L}$ corresponds to and-term in $\mathcal{G}$; $\rho \sim 0$ in $\mathcal{L}$ maps to rel-term in $\mathcal{G}$; the nonterminal ρ in $\mathcal{L}$ maps to the nonterminal exp in $\mathcal{G}$; and terminal r in $\mathcal{L}$ maps to const, and terminal $u(t)$ in $\mathcal{L}$ maps to cp in $\mathcal{G}$. The complete proof is in Appendix A.

Comparing $\mathcal{L}$ with Signal Temporal Logic (STL) [MN04], the logic $\mathcal{L}$ is able to express the temporal operator globally, i.e., G, from STL. Specifically, the STL property

$$G_{[0,500)}(th(t) < 100)$$

i.e., the throttle should remain less that 100 from 0s to 500s, corresponding to the following formula in $\mathcal{L}$:

$$\forall t \in [0, 500) : (th(t) < 100)$$

which corresponds to the following condition generated by the grammar $\mathcal{G}$ utilizing control points $c_{th,0}$, $c_{th,1}$, and $c_{th,2}$:

$$c_{th,0} < 100 \wedge c_{th,1} < 100 \wedge c_{th,2} < 100$$

In addition, $\mathcal{L}$ can express arithmetic operations within predicates, which are not part of the core STL formula syntax, thus extending STL with explicit arithmetic expression support. Specifically, the following formula in $\mathcal{L}$, $(\forall t \in [0, 100) : th(t) - p(t) \leq 20)$ cannot be directly specified in STL, as STL does not include arithmetic operators in its core syntax.

To demonstrate that $\mathcal{L}$ is capable of expressing common CPS properties, we assessed a dataset of 98 industrial CPS requirements previously used by Menghi et al. [MNGB19]. Menghi et al. formalized this dataset in restricted signal first-order logic, a logic fragment proposed in their study. Of the 98 formalized requirements, 85 can be expressed in our logic $\mathcal{L}$. The remaining 13 cannot, as they rely on existential quantifiers ($\exists$) or nested universal quantifiers ($\forall$), which lie outside the scope of $\mathcal{L}$. The fact that 86% of the

industrial CPS requirements can be expressed in $\mathcal{L}$ demonstrates that the logic fragment $\mathcal{L}$ remains highly expressive for capturing a wide range of real-world CPS properties. In our replication package [git25], we have included the 85 natural-language requirements along with their corresponding formalizations.

6.5 Evaluation

In this section, we evaluate GenTV using case studies from the CPS domain. Our case studies involve simulators and testbeds that are prone to flakiness, leading to potential variations in the datasets used to infer test validators. Our evaluation starts with RQ1, which assesses the extent of flakiness in the test results from our case-study systems. The goal is to confirm the presence of flakiness in these systems and to estimate its prevalence. In RQ2, we evaluate the accuracy and effectiveness of test validators generated by the alternative condition-inference techniques in Sections 6.3.1 and 6.3.2. In RQ3, we evaluate whether flakiness in the training sets affects the accuracy of test validators. When the SUT is flaky, re-running tests can produce different verdicts. Ideally, the generated test validators should remain robust, regardless of which run of the system the training set is derived from. In RQ4, we assess whether the test-validator assertions align with descriptions of precondition violations, ODD limits, and low-risk scenarios in technical standards and empirical studies.

RQ1 (Existence of Flakiness) *How flaky are our case-study systems?* We assess the level of flakiness in our network, ADS and aerospace case studies by calculating the

percentage of inconsistent test verdicts from multiple re-executions of randomly selected test inputs.

RQ2 (Accuracy) *How accurate are the assertion-based test validators inferred by our approach using different condition-inference methods?* We examine the accuracy of the assertion-based test validators generated by the alternative condition-inference techniques described in Sections 6.3.1 and 6.3.2.

RQ3 (Robustness to Flakiness) *How is the accuracy of test validator assertions impacted when using training sets from different executions of the SUT?* We assess whether flakiness in training sets impacts the accuracy of test validators. Since a flaky SUT can yield different outcomes across runs, the generated validators should remain robust regardless of which run the training data comes from. We examine the robustness of our assertion-generation technique to ensure its accuracy is consistent across different SUT executions.

RQ4 (Alignment) *To what extent do the assertion-based test validators align with violations of ODD limits, violations of preconditions, and low-risk scenarios in the real world?* We evaluate how closely the generated test validator assertions for our case studies align with the descriptions of precondition violations, ODD limits, and low-risk scenarios provided in the reference documentation. This documentation includes technical standards and other materials that define the operational principles and system requirements for the case studies. To address this research question, we translate a representative set of assertions from our experiments into natural language and assess their alignment with the corresponding reference documentation through a systematic human-subject study.

6.5.1 Case-Study Systems

We use the NTSS, autopilot, and ADS systems (introduced in Chapter 3) in our analysis. We provide more details below:

NTSS. Our first case-study system is the NTSS case study that we discussed in Section 3.3. For this case study, we employ SOHOSim, our testbed for NTSS. SOHOSim enables us to assess whether the user experience for streaming services is satisfactory (pass) or unsatisfactory (fail). Each test execution on NTSS takes approximately 4.5 minutes and is compute-intensive. In addition, there is non-determinism in the test results due to fluctuations in network bandwidth, latency, jitter, asynchrony in network flows, and the CPU and memory load on the machine hosting the testbed.

Aircraft autopilot system. We use the autopilot Simulink model from Table 3.1, which is derived from a public-domain benchmark of Simulink specifications provided by Lockheed Martin [loc25]. An example based on this system, simplified to have fewer inputs, is illustrated in Section 6.4.1. The Simulink model of autopilot captures both the autopilot system, which includes the control logic and algorithms responsible for stabilizing and navigating the aircraft, as well as the simulator, which simulates the aircraft's physical dynamics and environmental factors such as wind and turbulence. The inputs to the autopilot system are signals related to the flight dynamics of the aircraft such as throttle, pitch angle, turning rate, heading and desired flight objectives such as a target altitude. We encode these signals using control points and apply piecewise interpolation to connect the control points. The autopilot system is expected to satisfy the following system-level

requirement: when the autopilot is enabled, the aircraft should reach the desired altitude within 500 seconds in calm air.

The publicly available Simulink model of the autopilot system, provided by MathWorks, is developed in compliance with the DO-178C standard [do-25, aut25], which requires the software to respond predictably to the same inputs and conditions. Consequently, in this Simulink model, the gust amplitude and direction – despite being inherently stochastic – are fixed to specific values. Furthermore, the turbulence model uses fixed noise seeds to eliminate non-determinism. Thus, while the system's design incorporates stochastic elements, the publicly available model is intentionally made deterministic. In our experiments, we used this deterministic Simulink model of the autopilot system. The autopilot case study – while deterministic and thus not susceptible to flakiness – nonetheless involves complex signal-based inputs, making it an interesting system for evaluating the accuracy of the assertion-based test validator in RQ2.

ADS systems. Recall from Section 3.2 that we use two types of self-driving controllers as our ADS systems: (1) the autopilot controller of BeamNG and (2) DAVE2 [BdTD+16], a DNN model trained for end-to-end self-driving. To test the autopilot controller, we employ the two simulation environments discussed in Section 3.2: (1) a complex town map (TWN), and (2) a simpler environment with a two-lane road (SNG). We refer to the setup that tests the autopilot controller on the town map as AP–TWN, and the one testing it on the simpler road as AP–SNG. We test DAVE2 using only the simpler road map, as DAVE2 is specifically trained for this environment.

To determine whether a test passes or fails in our ADS setups, we consider the system-level requirements discussed in Section 3.2. Specifically, for AP–SNG and DAVE2, tested in the single-road environment, we consider requirement **R1** from Section 3.2. In the case of AP–TWN, operating in the complex town map, we consider requirements **R1**, **R2**, **R3** and **R4** from Section 3.2. Test execution time is approximately three minutes for AP–TWN and one minute for AP–SNG and DAVE2. Flakiness in these ADS test setups can arise due to inconsistencies in timing between the simulator and ADS controller, which may lead to variations in the images or sensory data received by the ADS. Furthermore, the addition of white noise to the images passed to the ADS may contribute to flakiness [ANN24]. In the more complex town map, factors like the presence of non-ego vehicles and traffic lights introduce additional flakiness in the test outcomes for AP–TWN.

Table 6.1 outlines the key characteristics of our case-study systems. These systems include the NTSS, the aircraft autopilot (AP–DHB), the ADS autopilot controller tested in a complete town (AP–TWN) and on a single-road map (AP–SNG), as well as the DNN-based controller tested on a single-road map (DAVE2). For AP–TWN, tested against the above-mentioned requirements (R1–R4), we present the results for each requirement separately.

6.5.2 RQ1 (Existence of Flakiness)

Experiment setting. RQ1 measures the degree of flakiness in our case studies, namely the NTSS, AP–TWN, AP–SNG, DAVE2, and AP-DHB. We randomly generate 100 test inputs for the NTSS case study and 200 test inputs for each of our ADS-based and autopilot

Table 6.1: Key characteristics of our case-study systems. *System* refers to the SUT aligned with the naming convention we adopt in the thesis. *Simulator* indicates the environment or testbed used for test execution. *Test Execution Time* indicates the approximate duration required to run a single test input on the corresponding simulator.

System	**Simulator**	**Test Execution Time**
NTSS (Section 3.3)	SOHOSim	~ 4.5 min
Aircraft autopilot system of De Havilland Beaver aircraft (AP–DHB) [aut25]	Simulink models of environmental factors (e.g., wind, turbulence, temperature, and atmospheric conditions) and aircraft's physical dynamics	~ 0.5 min
ADS autopilot tested in a complete town (AP–TWN)	BeamNG	~ 3 min
ADS autopilot tested on a single road map (AP–SNG)		~ 1 min
DNN self-driving controller tested on a single road map (DAVE2)		

case studies. Since the NTSS is our most resource-intensive system to execute, we generate a smaller number of test inputs for it. Each test input is executed 10 times to detect any non-determinism in the test outcomes. We then prepare ten datasets for each case study where each dataset contains the verdicts from a distinct execution of test inputs. We refer to each dataset as TS_n where n denotes the n-th execution of the test inputs. Consequently, for each case study, we obtain datasets TS_1, TS_2, ..., TS_{10}.

Results. Table 6.2 shows the percentage of flaky tests observed for each case study across TS_1 to TS_{10}. We consider a test case to be flaky unless all ten runs produce the same outcome. The percentage of flaky tests for AP–DHB is 0%, indicating the absence of flakiness in this system. In contrast, the percentage of flaky tests for NTSS is 11%. The percentage of flakiness for the tests exercising AP–TWN for requirements R1 to R4 ranges between 21% and 79%.

Table 6.2: Percentage of flaky tests in the systems of Table 6.1

NTSS	AP–DHB	AP–TWN				AP–SNG	Dave2
		R1	R2	R3	R4		
11%	0%	79%	64%	70%	21%	1.5 %	33%

Finding. Except for AP–DHB, all our case-study systems exhibit flaky tests, with rates ranging from 1.5% to 79%, demonstrating their susceptibility to flakiness.

6.5.3 RQ2 (Accuracy)

Experiment setting. To answer RQ2, we generate test validators using the GP, DT, and DR alternatives by applying these methods to the training sets for each case study. Specifically, for each case study, we select one of the ten datasets from RQ1 to serve as the training set. This enables us to assess each method without regard to the variations caused by flakiness across the different datasets from RQ1. Analyzing the impact of the variations caused by flakiness is left for RQ3. We tune the hyper-parameters of DT and DR using Bayesian Optimization [SLA12]. We configure GP using the parameters in Table 6.3 and apply GP with each of the fitness functions from Section 2.3: Naish denoted by GP_N, Tarantula denoted by GP_T, and Ochiai denoted by GP_O. To account for the randomness of GP, DT, and DR, we apply each technique 20 times to the training set for each case study.

In addition to considering the test validator generation methods individually, we also consider an *ensemble* approach. Specifically, for each run of GP_N, GP_T, GP_O, DT, and DR, the ensemble method computes the union of the conditions generated by these tech-

Table 6.3: Parameters of GP: mutation rate (*Mut_rate*), crossover rate (*Cr_rate*), population size (*Pop_size*), number of generations (*Num_gen*), max. tree depth (*Max_d*), tournament size (*T_size*), split Criterion (*Criterion*), split strategy (*Splitter*), min. samples to split a node (*Min_split*), min. samples per leaf (*Min_sample*)

Parameter	**Value**	**Parameter**	**Value**	**Parameter**	**Value**	**Parameter**	**Value**
Mut_rate	0.1	Num_gen	50	Pop_size	50	Max_const	ADS: 100 NTSS: 400 Autopilot: 45
Cr_rate	0.7	T_size	7	Max_d	5	Min_const	ADS: 0 NTSS: 0 Autopilot: -30

Note: The values within the framed box are from [LNS24, GMN+21, LP06]. The values within the framed box are set by assessing the average number of generations required to reach a plateau. The values within the framed box are based on the lowest and highest values that the input variables of our systems can assume.

niques. Then, we derive a consistent set of assertions by applying the third step of GenTV (Section 6.3.3) to this union. We then compare the assertion-based test validators generated by the ensemble with those generated by each method individually.

Recall from Definition 6.2 that each SUT is associated with a user-defined verdict threshold θ, which specifies the minimum confidence level required for an assertion to be included in the test validator. We vary the user-defined verdict threshold θ, from 0.5 to 1 in increments of 0.05. We do not consider $\theta < 0.5$, since verdict predictions by assertions with less than 50% confidence are unlikely to be trusted and used in practice.

We generate each case study's test set by randomly creating inputs and executing them on the case-study system to obtain ground-truth verdicts. To mitigate the impact of flaky behaviour on the ground-truth verdicts for test sets, we do as follows:

- For systems with a flaky test rate below 50%, we include only tests that exhibit consistent behaviour in all ten runs. Any test that shows flakiness in those runs is excluded from the test set.
- For systems with a flaky test rate above 50%, since completely excluding flaky tests is cost-prohibitive, we include tests that produce consistent verdicts in at least eight out of ten runs. Their final verdicts are determined by majority voting.

We ensure that each test set for each case study ultimately contained 200 elements.

Results. To answer RQ2, we assess the accuracy of the generated test validators using the metrics from Section 6.2.3: accuracy, misprediction rates – i.e., the rate of pass verdicts predicted as fail, and the rate of fail verdicts predicted as pass – and relative accuracy. In addition to these metrics, we compare the test validator generation methods based on the percentage of *unique correct predictions*. Specifically, given a pair of methods A and B and a test set *TestSet*, the percentage of unique correct predictions for method A is the percentage of tests in *TestSet* whose verdicts are correctly predicted by A but not by B. Figure 6.9 illustrates using a Venn diagram notation the percentage of unique correct predictions for methods A and B as the "only A" and "only B" areas, respectively. The percentage of unique correct predictions indicates how well a method complements other methods. Below, we assess different test validator generation methods using accuracy, misprediction rates, relative accuracy, and the percentage of unique correct predictions.

All statistical tests are performed using the Mann-Whitney U test [MW47] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [VD00]. All statistical significance tests in RQ2 are reported with p-values adjusted using the Benjamini–Hochberg (BH) procedure [BH95]. We

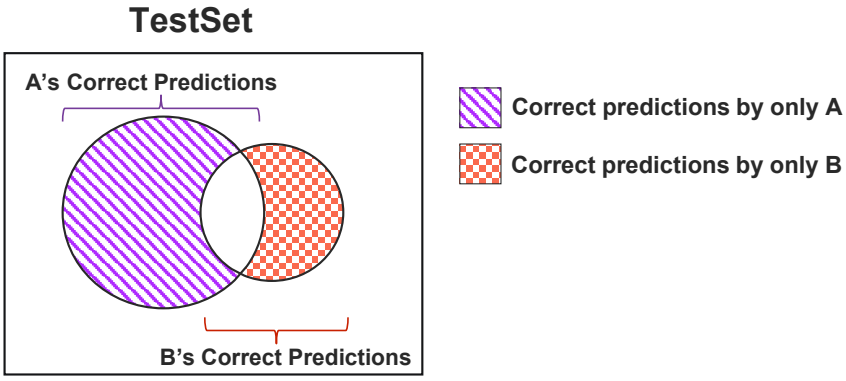


Figure 6.9: Illustrations of the percentage of unique correct predictions on a *TestSet* made by methods A and B

classify effect size values for accuracy and relative accuracy, where higher values indicate better performance, as follows: effect sizes are classified as small, medium, and large when their values are greater than or equal to 0.56, 0.64, and 0.71, respectively [VD00]. For misprediction rates, where lower values indicate better performance, effect sizes are classified as small, medium, and large when their values are lower than or equal to 0.44, 0.36, and 0.29, respectively [VD00].

(1) Accuracy. Figure 6.10 shows the average accuracy of the test validators generated by each technique for all case studies when θ varies from 0.5 to 1. The average accuracy of test validators generated by GP_T, GP_O, GP_N and ensemble decreases as θ increases because a higher value of θ results in fewer assertions in these test validators, as we only retain those with a confidence level of at least θ. In contrast, since the confidence levels of assertions produced by DT and DR are generally high (i.e., above 0.8), the accuracy of DT and DR remains relatively stable as θ increases.

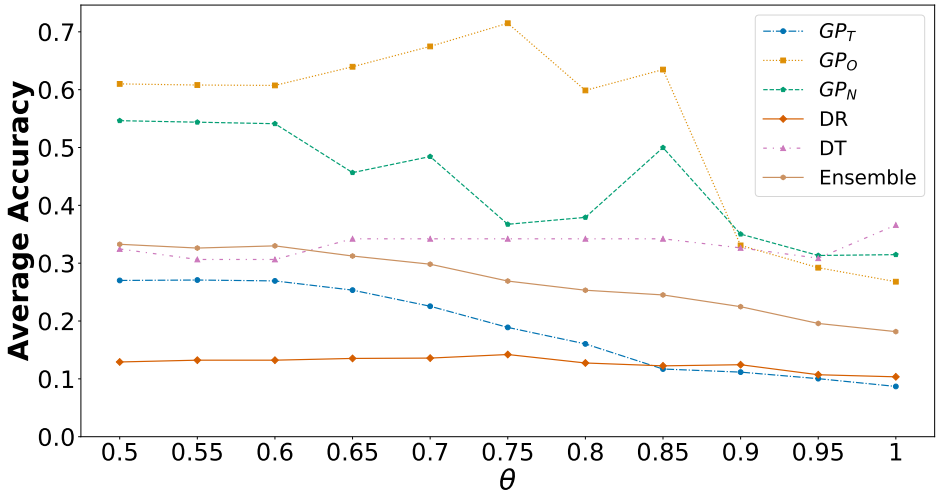


Figure 6.10: Average accuracies of the test validators generated by GP_T, GP_O, GP_N, DT, DR and ensemble for all case studies when the verdict threshold θ varies from 0.5 to 1.

For $0.5 \leq \theta \leq 0.85$, GP_O produces the most accurate test validators compared to other techniques. Statistical tests comparing the accuracy results in Figure 6.10 are provided in Table B.1 in Appendix B. Based on this table, for $0.5 \leq \theta \leq 0.85$, test validators generated by GP_O are significantly more accurate than those generated by other techniques. The effect-size values for the comparisons of GP_O with GP_T, DT, DR, and the ensemble are all large, while the comparisons of GP_O with GP_N show both large and small effect sizes. For $\theta \geq 0.9$, there are no statistically significant differences in accuracy between GP_O and GP_N, DT or the ensemble method.

Statistical tests comparing the accuracy results of GP_O with GP_T, GP_N, DT, DR, and the ensemble for each case-study system across all values of θ are provided in Table B.2 in Appendix B. Based on these results, test validators from GP_O are significantly more

accurate than those from GP_T and DR in all eight systems, outperform ensemble in seven, DT in six, and GP_N in five of the eight case studies. The effect-size values for these comparisons are small, medium or large.

For AP–TWN (R4), test validators generated by DT fail to predict any verdicts across all values of θ. This is shown in Table B.2 by stating *DT is not applicable* and highlighting the corresponding cells in yellow. We do not show the comparisons of GP_T, GP_N, DT, DR and ensemble with each method, as these comparisons provide no additional insights beyond Figure 6.10 and Tables B.1 and B.2. Full comparisons are available in our supplementary material [git25].

Finding. For $0.5 \leq \theta \leq 0.85$, test validators generated by GP with Ochiai achieve significantly better accuracy than other methods, outperforming the interpretable ML models (DT and DR) by at least 25% in terms of accuracy on average. In contrast, for $\theta \geq 0.9$, there is no statistically significant difference in accuracy between GP with Ochiai and Naish, DT, or the ensemble method.

(2) Misprediction Rates. Figures 6.11 and 6.12 show the rate of actual pass verdicts predicted as fail, and the rate of actual fail verdicts predicted as pass, respectively, for all case studies when θ varies from 0.5 to 1. For clarity, we report these two rates across the following three aggregated θ-ranges, since presenting them for each individual θ does not provide a concise overview: low ($0.5 \leq \theta < 0.7$), medium ($0.7 \leq \theta < 0.9$), and high ($0.9 \leq \theta$). We refer to the rate of pass verdicts predicted as fail as *Pass-as-Fail*, and the rate of fail verdicts predicted as pass as *Fail-as-Pass*.

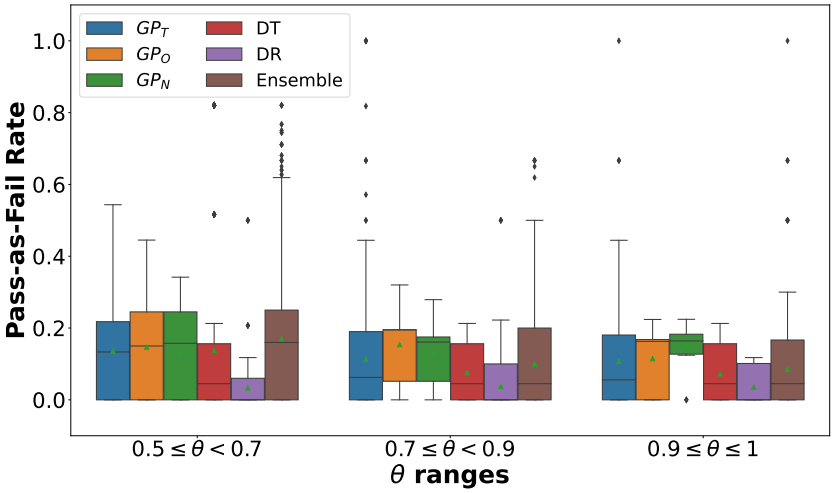


Figure 6.11: Pass-as-Fail rates of the test validators generated by GP_T, GP_O, GP_N, DT, DR and ensemble for all case studies when the verdict threshold θ varies from 0.5 to 1.

As shown in Figure 6.11, across all θ ranges, DR consistently achieves the lowest (best) average Pass-as-Fail rates compared to all GP techniques – GP_T, GP_O, and GP_N – as well as the ensemble and DT. In contrast, for the Fail-as-Pass misprediction results, the GP techniques overall, and in particular GP_O, produce better results compared to DT, DR, and the ensemble. Statistical tests comparing the Pass-as-Fail (Figure 6.11) and Fail-as-Pass (Figure 6.12) results are provided in Tables B.3(a) and (b), respectively, in Appendix B. Specifically, for the Pass-as-Fail rate, we compare the best-performing method for this metric, DR, with the other techniques, and for the Fail-as-Pass rate, we compare its best-performing method, GP_O, with the others. Based on Table B.3(a), test validators generated by DR lead to a significantly lower rate of Pass-as-Fail compared to those obtained by other techniques with small, medium or large effect sizes. Based on Table B.3(b) for $0.5 \leq \theta < 0.9$, GP_O either achieves a significantly lower Fail-as-Pass rate (with small, medium or large

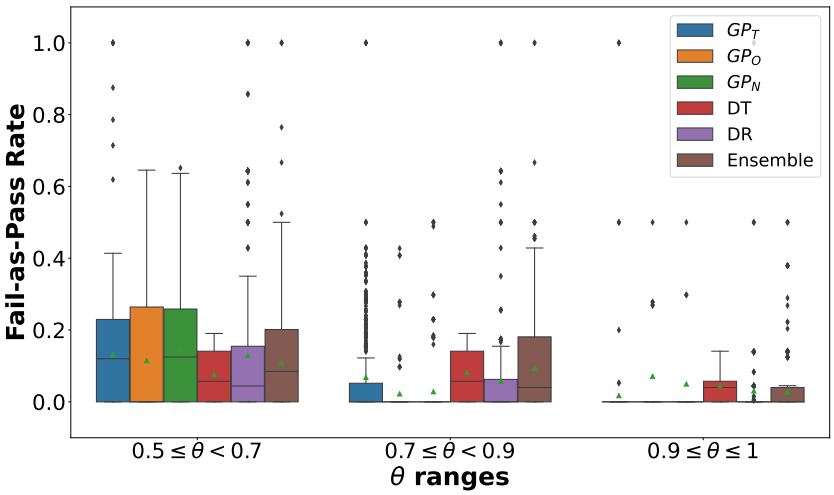


Figure 6.12: Fail-as-Pass rates of the test validators generated by GP_T, GP_O, GP_N, DT, DR and ensemble for all case studies when the verdict threshold θ varies from 0.5 to 1.

effect sizes) or shows no statistically significant difference in Fail-as-Pass rate compared to other techniques. For $0.9 \leq \theta \leq 1$, GP_T results in a significantly lower Fail-as-Pass rate compared to other techniques.

Table B.4 in Appendix B presents the statistical tests comparing the Pass-as-Fail rate of DR and the Fail-as-Pass rate of GP_O with those of the other techniques, for each case-study system and across all values of θ. Based on Table B.4(a), test validators generated by DR have a significantly lower Pass-as-Fail rate than those obtained by GP_N in all eight case-study systems, by GP_T and GP_O in seven, by ensemble in six, and by DT in four out of eight case-study systems. Based on Table B.4(b), test validators generated by GP_O have a significantly lower Fail-as-Pass rate than those obtained by DR in five case-study

systems, and by GP_T and ensemble in four case-study systems. The effect-size values in all the comparisons are negligible, small, medium or large.

Finding. Test validators generated by DR achieve a significantly lower Pass-as-Fail rate than those of other methods across all verdict thresholds θ between 0.5 and 1, outperforming GP-based techniques (GP with Tarantula, Ochiai, Naish) by at least 6% on average. In contrast, for $0.5 \leq \theta \leq 1$, test validators generated by GP with Ochiai either result in a significantly lower Fail-as-Pass rate compared to interpretable ML models (DT and DR) and the ensemble method or exhibit no statistically significant difference in Fail-as-Pass rate compared to these techniques.

(3) Relative Accuracy. Figure 6.13 presents the average *relative* accuracy of the test validators generated by GP, DT, DR and ensemble for all our case studies and for $0.5 \leq \theta \leq 1$. Recall from Section 6.4 that while accuracy is the percentage of correct predictions among all predictions, relative accuracy is the percentage of correct predictions excluding inconclusive predictions. Based on Figure 6.13, the average relative accuracies of GP, DT and DR, for all values of θ, exceed 0.7, indicating that all the compared methods have high levels of correctness when they make conclusive predictions.

The test validators generated by DR show consistently higher average relative accuracy compared to the other techniques across all values of θ, with the exception of $\theta = 0.7$, where test validators generated by GP_O achieve a higher average of relative accuracy than those of DR. This superior relative accuracy over GP techniques – GP_T, GP_O and GP_N – is because DR generates stronger assertions containing multiple logical terms. In contrast, GP

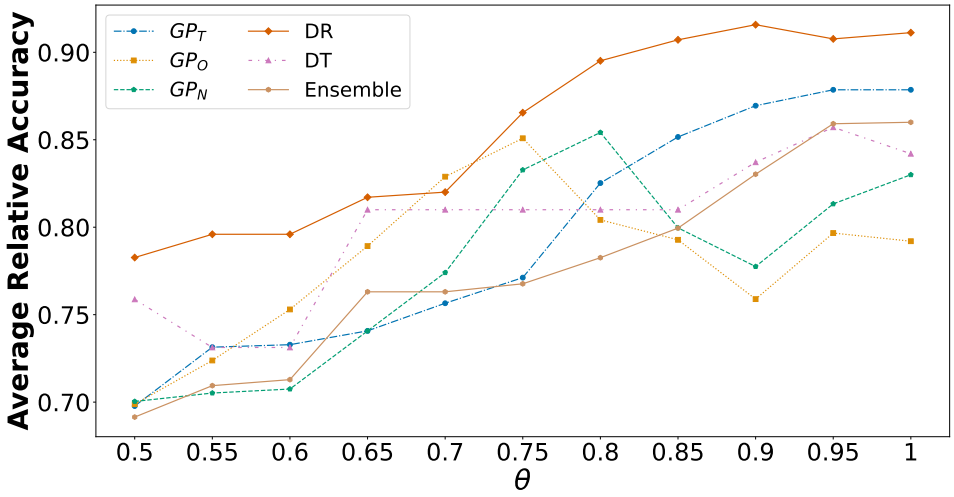


Figure 6.13: Average relative accuracies of the test validators generated by GP_T, GP_O, GP_N, DT, DR and ensemble for all case studies when the verdict threshold θ varies from 0.5 to 1.

techniques generate assertions with fewer logical terms. The weaker assertions generated by GP techniques can provide predictions for more test inputs compared to DR. However, the higher number of predictions made by GP techniques also increases their susceptibility to mispredictions compared to DR.

Statistical tests comparing the relative accuracy results in Figure 6.13 are in Table B.5 in Appendix B. These statistical tests are consistent with the average comparisons discussed above. Specifically, test validators generated by DR lead to significantly higher relative accuracy compared to those obtained by the other techniques across all values of θ, except for $\theta = 0.7$ and $\theta = 0.95$, where no statistically significant differences in relative accuracy are observed between DR and GP_O, and between DR and GP_T, respectively.

Table B.6 in Appendix B presents the statistical tests comparing the relative accuracy of DR with GP_T, GP_O, GP_N, DT and the ensemble for each case-study system across all values of θ. Based on this table, test validators generated by DR lead to a significantly higher relative accuracy than those generated by GP_O and GP_N in all eight case-study systems, by GP_T and DT in six, and by ensemble in five case-study systems. For the router case study, test validators generated by GP_T and the ensemble yield significantly higher relative accuracy compared to DR.

Finding. For most values of the verdict threshold θ, test validators generated by DR achieve significantly higher relative accuracy compared to those obtained by other techniques.

(4) Unique Correct Predictions. We compare DR, which performs best in relative accuracy and Pass-as-Fail rate, with GP_O, which performs best in accuracy and Fail-as-Pass rate, based on the percentage of unique correct predictions made by each method. Table 6.4(a) presents the results of this comparison for all values of θ across all case-study systems and Table 6.4(b) shows the results for all case-study systems across all values of θ.

Based on Table 6.4(a), on average, the percentage of unique correct predictions for GP_O is higher than the percentage of unique correct predictions for DR (20% for GP_O versus 6% for DR). For $0.5 \leq \theta < 0.9$, the percentage of unique correct predictions for GP_O is significantly higher than the percentage of unique correct predictions for DR, with a large effect size. In contrast, for $0.9 \leq \theta \leq 1$, the percentage of unique correct predictions

Table 6.4: Percentage of unique correct predictions by GP_O and DR.

(a) Comparison for different values of verdict threshold θ across all study subjects

θ	GP_O: **Correct Prediction** **DR: Incorrect or Inconclusive Prediction**	**DR: Correct Prediction** GP_O: **Incorrect or Inconclusive Prediction**
0.5	42%	5%
0.55	41%	5%
0.6	35%	6%
0.65	30%	6%
0.7	25%	6%
0.75	19%	7%
0.8	11%	7%
0.85	9%	6%
0.9	2%	8%
0.95	1%	5%
1	1%	4%
Average	20%	6%

(b) Comparison for each study subject across all values of the verdict threshold θ

Study Subject	GP_O: **Correct Prediction** **DR: Incorrect or Inconclusive Prediction**	**DR: Correct Prediction** GP_O: **Incorrect or Inconclusive Prediction**
NTSS	27%	7%
AP–DHB	10%	2%
AP–TWN (R1)	4%	8%
AP–TWN (R2)	15%	7%
AP–TWN (R3)	36%	4%
AP–TWN (R4)	22%	~ 0%
AP–SNG	32%	5%
Dave2	8%	18%

for DR is significantly higher than the percentage of unique correct predictions for GP_O with a large effect size. In addition, based on Table 6.4(b), GP_O achieves a higher average percentage of unique correct predictions than DR for six out of eight case-study systems.

Finding. Test validators generated by GP with Ochiai correctly predict, on average, 14% more unique tests than those of DR.

▶ Size of assertions GP_O, and GP techniques in general, generate weaker assertions, characterized by a smaller number of relational terms. We measure the length of the generated assertions as the number of relational terms they contain. Assertions generated by GP techniques contain, on average, 2.1 relational terms involving one or two input variables. This pattern is consistent across both the autopilot case study, which has seven input variables, and the ADS case studies, which have eight input variables. In the router system, where assertions often involve arithmetic combinations of data flows, GP-generated assertions include summations over up to four of the eight input variables. For the autopilot and ADS case studies, DT-generated assertions contain, on average, 3.5 relational terms involving up to three input variables, while DR-generated assertions contain, on average, 2.3 relational terms involving up to two input variables. For the router system, DT-generated assertions contain, on average, 3.2 relational terms, involving summations over up to three of the eight input variables, while DR-generated assertions include, on average, 2.1 relational terms, with summations over at most three input variables. Across all case-study systems, the generated assertions, include no more than five relational terms.

▶ Recommendation on setting θ While RQ2 compares different methods for assertion generation with θ varying between 0.5 and 1, when strong assurance of verdict prediction is required, we expect θ values higher than 0.9 to be considered. For $\theta \geq 0.9$, GenTV becomes inconclusive for many tests. However, due to the following two reasons, even with a moderate rate of conclusive verdicts, GenTV remains useful:

(1) *Conclusive predictions by our test validators at high verdict thresholds are most accurate, with a negligible rate of predicting actual failures as passes.* In particular, for our case-study systems, the rate of Fail-as-Pass is zero or negligible ($< 3\%$) when

$\theta \geq 0.9$. This is particularly important in safety-critical systems, where minimizing the risk of misclassifying a failing test as passing is essential to avoid overlooking potential failures.

(2) *Even a moderate rate of conclusive verdicts yields substantial execution-time savings.* Given the high accuracy rate and the negligible incidence of Fail-as-Pass, even a moderate proportion of conclusive verdicts is valuable. For instance, for $\theta \geq 0.9$, test validators generated by GenTV for checking car accidents with other vehicles (i.e., the AP–TWN(R2) case study) generate conclusive verdicts for 30% of test cases, thus avoiding execution for these test cases. Given that ADS simulations have a high flakiness rate and are time-consuming (each simulation takes at least three minutes based on Table 6.1), avoiding execution for 30% of tests represents significant time savings. For checking lane evasions (i.e., the AP-SNG case study), when $\theta \geq 0.9$, our test validators produce conclusive verdicts for 80% of test cases, meaning that only 20% of tests require execution. For evaluating users' quality of experience in the router system, when $\theta \geq 0.9$, our test validators produce conclusive verdicts for 15% to 60% of test cases, depending on the underlying technique used in GenTV for generating test validators. Given the router system's flakiness and the time-intensive nature of its simulations (each taking at least 4.5 minutes based on Table 6.1), avoiding 15% to 60% of test executions results in significant time savings.

6.5.4 RQ3 (Robustness to Flakiness)

Experiment setting. For RQ3, we use the ten datasets, TS_1 to TS_{10}, from RQ1 for each flaky case-study system, i.e., NTSS, AP–TWN (R1) to (R4), AP–SNG and DAVE2. As noted in RQ1, while these datasets contain identical test inputs for a given system, the verdicts assigned to these inputs may vary due to flakiness. In RQ3, we assess the robustness of each assertion-inference technique across these ten datasets for each case-study system. Specifically, we apply the GP_T, GP_O, GP_N, DT, DR, and ensemble methods to each dataset to generate assertion-based test validators. Each technique is configured using the same parameters as in RQ2, and similarly to RQ2, we apply each technique 20 times to each dataset to account for randomness.

For each case study, we use the same test sets generated in RQ2 to measure the accuracy of the generated test validators. To assess variations in the accuracy of test validators, we report the average absolute deviation (AAD) of accuracy values for each case study. Specifically, given a distribution of accuracy values, AAD is computed as the average deviation of individual accuracy values from the mean accuracy. A low AAD indicates that the test validators' accuracy is less impacted by variations in verdicts across the different training datasets.

All statistical tests are performed using the Mann-Whitney U test and the Vargha-Delaney's effect size. All statistical significance tests in RQ3 are reported with p-values adjusted using BH procedure [BH95].

Results. Table 6.5 shows the average accuracy of the test validators for all verdict thresholds θ and computed using the ten datasets, TS_1 to TS_{10}, generated in RQ1. Based

Table 6.5: Average accuracy of GP_T, GP_O, GP_N, DT, DR and ensemble when datasets TS_1 to TS_{10} are used for each study subject in RQ3. The cells highlighted in blue represent the maximum average accuracy obtained for each case study.

	NTSS	**AP–TWN**				**AP–SNG**	**Dave2**	***Average***
		R1	**R2**	**R3**	**R4**			
GP_T	6%	8%	13%	17%	9%	8%	16%	11%
GP_O	57%	60%	41%	97%	54%	81%	37%	60%
GP_N	40%	44%	37%	45%	36%	42%	48%	42%
DT	24%	19%	34%	23%	15%	18%	19%	22%
DR	14%	8%	10%	8%	1%	12%	22%	11%
Ensemble	23%	10%	25%	19%	6%	17%	28%	18%

on this table, the average accuracy of the test validators generated by GP_O surpasses that of other techniques across all case studies, except for DAVE2, where the test validators generated by GP_N achieve the highest average accuracy.

Statistical tests comparing the accuracy results in Table 6.5 for each verdict threshold θ are provided in Table B.7 in Appendix B. As shown in this table, test validators generated by GP_O lead to significantly higher accuracy compared to those obtained by other techniques across all values of θ, except for $\theta = 1$ (i.e., 100%), where no statistically significant differences in accuracy are observed between GP_O and GP_N.

Table 6.6 presents the AAD in the accuracy of test validators generated based on the TS_1 to TS_{10} datasets for our case studies. Based on this table, variations in the TS_1 to TS_{10} datasets result in an average fluctuation of 4% in the accuracy of test validators generated by GP_O. In contrast, validators produced by GP_T, GP_N, and ensemble exhibit larger fluctuations. Similarly, test validators generated by DT and DR have accuracy fluctuations of 5% and 3%, respectively, indicating they are less affected by flakiness in the datasets. However, as Tables 6.5 and B.7 show, these test validators achieve lower accuracy

Table 6.6: Average Absolute Deviation (AAD) of accuracy of GP_T, GP_O, GP_N, DT, DR and ensemble for ten datasets generated by executing a set of test inputs ten times for our case studies in RQ3

	NTSS	**AP–TWN**				**AP–SNG**	**Dave2**	***Average***
		R1	**R2**	**R3**	**R4**			
GP_T	7%	4%	9%	10%	5%	7%	9%	7%
GP_O	7%	5%	10%	0%	4%	0%	4%	4%
GP_N	13%	12%	10%	13%	9%	13%	7%	11%
DT	15%	4%	6%	3%	2%	1%	4%	5%
DR	9%	3%	1%	1%	2%	1%	6%	3%
Ensemble	17%	7%	10%	12%	7%	2%	7%	9%

than those generated by GP_O. Thus, GP_O produces the most accurate and robust test validators.

Finding. Overall, our results indicate that test validators generated by GP with Ochiai are the most accurate and robust compared to those produced by other techniques.

Take away 1. SBFL ranking formulas integrated with GP generate accurate assertion-based test validators. In particular, Ochiai is well-suited for generating accurate test validators that are robust against flakiness.

Take away 2. When GP is used with Ochiai, flaky tests in the training set have only a negligible impact on the accuracy of the inferred test validators. Hence, removing flaky tests does not significantly alter the validators' accuracy. Therefore, practitioners can save effort when preparing training sets for test validator generation, as excluding flaky tests is not critical.

Table 6.7: Reference documentation for case-study systems

System	Type of Reference	Documentation
NTSS	Industry standard	Traffic-shaping manual [HJTM18, CAK25] ITU-T Recommendation E.361 [ITU25]
ADS	Industry standard	SAE J3016 [Pra21]
	Published empirical and expert-validated results	[YSYA19, CWX24, LSWH24, WAYZ25, WAWZ25, BNPR20, BES+23, ANBS18]
AP-DHB	Industry standard	Autopilot system handbooks [aut25, Adm09]

6.5.5 RQ4 (Alignment)

Experiment setting. We evaluate how closely the generated test validator assertions for our case studies align with the descriptions of precondition violations, ODD limits, and low-risk scenarios provided in the reference documentation for these case studies. Table 6.7 lists the reference documentation considered for each case-study systems: For NTSS, our sources are the technical standard on priority-based flow management [HJTM18, CAK25] and ITU-T Recommendation E.361 [ITU25], which specifies network quality-of-service (QoS) parameters and performance requirements. For ADS, we rely on the SAE J3016 standard [Pra21] – which defines the ADS taxonomy and the human driver's role at each automation level – along with published ADS-related results based on simulations, field tests, and other expert-validated findings [YSYA19, CWX24, LSWH24, WAYZ25, WAWZ25, BNPR20, BES+23, ANBS18]. For AP–DHB, we draw on autopilot system handbooks [(AS09, Adm09], which describe operational principles, performance limits, and safety constraints.

From the test validators generated by GenTV in RQ2, we select a subset of assertions for each case study. We consider high-confidence assertions (i.e., $\theta \geq 0.9$), following our conclusion from RQ2 that these yield the highest accuracy and the lowest rate of mis-predicting failures as passes. We apply stratified sampling [SM96] to each of the eight case studies, selecting 20 high-confidence assertions per system (160 in total), stratified by their binary outcome labels (pass vs. fail), to ensure balanced representation of both categories. We develop textual assertions and evaluate their alignment with the reference documentation through a systematic human-subject study[1], as described in the following three steps:

Step 1: Translate logical assertions into natural language. We translate logical and arithmetic expressions into words, and map variables and constants to their domain-specific meanings using terminology from the reference documentation (e.g., $pitchwheel(t)$ is translated into "the angle of the nose of the aircraft"). The translations were collaboratively produced by the authors, led by the first author, and jointly reviewed to ensure accuracy and consistency. Examples of these translations are shown in Table 6.8, and the complete list of assertions, along with their translations, is in our supplementary material [git25].

Step 2: Identify sentences in the reference documentation that are related to each assertion. For each textual assertion, we use OpenAI's API [Ope25] (GPT-5, medium reasoning, with a context window of 400k tokens) to find semantically related sentences in the system's reference documentation. Using the template in Figure C.1 (Appendix C), we construct one-shot prompts – one per natural-language assertion – each containing (1)

[1]Written informed consent was obtained from all participants prior to their participation in the study. Ethics approval for this human-subject study was granted by the University of Ottawa's Research Ethics Board (file number H-11-25-12283).

Table 6.8: Assertion examples for AP-DHB, NTSS, AP-TWN (R2), DAVE2 and AP-SNG case studies along with their natural-language translation, sentences retrieved automatically by GPT as being related to the textual assertions, and labels from human annotators. Segments highlighted in green show where the translated assertions align with the retrieved sentences, while segments highlighted in orange in the example assertion labelled as overlapping mark information that is missing from the assertion but present in the retrieved sentences.

Case Study	**Assertion**	**Assertions stated in Natural Language**	**Retrieved Sentences**	**Label**
AP–DHB	$(\forall t \in [0,500) : p(t) \leq 0) \wedge (\forall t \in [0,250) : th(t) \leq 0.3) \Rightarrow fail$	If the nose of the aircraft is pointed downwards for 500 seconds and the thrust applied to the engine is low (less than 30% of maximum thrust) for 250 seconds, the aircraft will not reach the required altitude in 500 seconds.	"Consequently, the tail is again pushed downward and the nose rises into a climbing attitude" [Adm09]. "If thrust decreases and airspeed decreases, lift will become less than weight and the aircraft will start to descend" [Adm09]. "Climb performance is directly dependent upon the ability to produce either excess thrust or excess power" [Adm09].	Aligned
Router	$flow_5 + flow_6 + flow_7 > 372 \Rightarrow fail$	Attempting to use high-priority DiffServ classes (classes 5, 6, and 7) so that they jointly exceed 68% of their allocated bandwidth share (more than 372 mb/s) degrades the quality of experience across the network.	"Secondly, it shows that the high-priority flows are limited so as to not use more than the share of the bandwidth allocated to the high-priority DiffServ classes" [HJTM18].	Aligned
AP–TWN (R2)	$weather = foggy \wedge time\_of\_day = night \wedge initial\_speed > 90 \wedge traffic\_density = high \Rightarrow fail$	While travelling at a high speed (more than 90 km/h) through a town on a foggy night, the ego-vehicle collides with nearby vehicles in dense traffic.	"Visible light camera is vulnerable to bad conditions such as fog and are difficult to see without a light source at night" [YSYA19]. "According to reports that evaluated the measurement distance, it was confirmed that the LiDAR measurement distance decreased as fog became darker, and the visibility distance became shorter" [YSYA19].	Aligned
DAVE2	$weather = sunny \wedge initial\_speed < 10 \Rightarrow pass$	While travelling at a low speed (less than 10 km/h) on a sunny day, the ego-vehicle keeps its lane.	"Both test vehicles were able to maintain lane keeping during all runs under Baseline conditions (ambient air temperatures between 20 °F and 100 °F, peak wind speeds below 22.4 mph, sun position greater than 15 ° above the horizon, ambient daylight conditions with clear sky, and dry and clear pavement)" [BES+23].	Overlapping
AP–SNG	$initial\_speed < 6 \wedge road\_angle \leq 5 \wedge initial\_position = centre\_of\_lane \Rightarrow pass$	When the ego vehicle is positioned at the center of the lane, is driving very slowly (less than 6 km/h), and the road is straight or slightly curved, the ego vehicle keeps its lane.	"Crash risk is minimal when a vehicle travels along the centerline of a lane, and the vehicle heading is parallel to the centerline" [CWX24]. "Lane-keeping-assist driving remains centered when the oncoming vehicle approaches the self-driving car, and that the human driver actively avoids the oncoming vehicle" [WAWZ25].	Unrelated (but judged as plausible safe or low-risk scenario)

the assertion and (2) system reference documentation related to that assertion, provided in a vector database. Each prompt instructs GPT to return two to five of the most related sentences for the assertion. We ran the prompt twice for each assertion. For approximately 90% of the assertions, the identified sentences were identical across the two runs. For the remaining assertions, we took the intersection of the identified sentences as the final set of related sentences. No assertion yielded an empty intersection. Finally, we manually verified all selected sentences to ensure that they appeared verbatim in the documentation and were not hallucinations.

Step 3: Evaluate the extent to which each assertion aligns with the retrieved sentences in Step 2. To evaluate how closely assertions align with the retrieved sentences in Step 2, we collaborated with two third-party annotators (non-authors). Both are graduate students in computer science with over two years of experience in software testing and requirements analysis, and have conducted research on CPS, ADS, and Simulink models. Among the pool of potential participants, these two candidates were selected because their background most closely matched the required domain expertise. Annotators were provided with textual assertions along with the retrieved sentences for each assertion as well as the following four-point Likert scale: (1) *Aligned*: The situation described by the assertion either (a) matches the situation described in at least one retrieved sentence, or (b) is a more specific instance of it (i.e., the assertion stays entirely within the scope of the retrieved sentences and does not extend beyond what they entail). Further, the assertion is not inconsistent with any of the retrieved sentences. (2) *Overlapping*: The situation described by the assertion is strictly broader in some respect and only partially overlaps with the situations described in at least one retrieved sentence (i.e., the assertion generalizes the

retrieved sentences or covers cases not supported by the retrieved sentences alone). Further, the assertion is not inconsistent with any of the retrieved sentences. (3) *Inconsistent*: The assertion is inconsistent with at least one of the retrieved sentences. (4) *Unrelated*: The assertion and the retrieved sentences are not aligned, overlapping or inconsistent. For these assertions, we asked the annotators to judge whether the situation described by the assertion corresponds to (i) a safe or low-risk situation, (ii) a violation of the ODD or preconditions, or (iii) neither.

We divided the 160 assertions (8 case-study systems $\times$ 20 assertions per system) equally between the two annotators, with a 20% overlap. As a result, each annotator evaluated 96 assertions in total. Assertions were randomly assigned and balanced across the case-study systems. We then conducted a three-hour training session to calibrate the annotators on the Likert scale, clarify labelling criteria, and ensure a consistent understanding of the annotation task. During the training, after introducing the Likert scale and discussing illustrative examples outside our experimental materials, we asked both annotators to independently label the 20% overlapping assertions. We then calculated the disagreement rate, which was approximately 9%, corresponding to a Cohen's kappa (κ) value of 0.75, indicating substantial agreement [Coh60]. A disagreement was counted whenever the two annotators assigned different Likert labels to the same assertion. All disagreements occurred when one annotator labelled an assertion as aligned and the other as overlapping. We subsequently discussed and resolved all disagreements with both annotators. At the end of the training meeting, after reaching consensus on the disagreements, both annotators felt confident in annotating their respective sets of non-overlapping assertions. The annotators then independently labelled the remaining assertions.

Table 6.8 shows examples of aligned, overlapping, and unrelated assertions based on labels from the annotators. For AP–DHB, the assertion that the aircraft's nose points downward while the engine operates at low thrust is *aligned* with the retrieved sentences, which state that the nose must rise to a climbing attitude or that excess thrust is required to gain altitude. For DAVE2, the assertion that the vehicle keeps its lane when driving at low speed on a sunny day is *overlapping*, because the first retrieved sentence includes specific conditions regarding air temperature, wind speeds, sun position, and clear pavement in addition to daylight conditions and a clear sky, whereas the assertion omits these details and therefore goes beyond what that sentence specifies. For AP–SNG, the assertion is labeled as *unrelated*, but it was nevertheless rated by our annotators as a plausible, low-risk lane-keeping situation. We note that no assertions in our study were labelled inconsistent by the annotators.

Results. Table 6.9 shows, for each case-study system, the percentage of aligned, overlapping, and unrelated assertions among all the selected assertions for this research question. At least 75% of the assertions for each system are either aligned or overlapping with the reference documentation. This high rate of alignment indicates that the generated assertions effectively capture situations that violate environmental assumptions or preconditions, describe low-risk operational states, and identify conditions that exceed the system's ODD limits.

Finding. On average, 80% of the high-confidence assertions ($\theta \geq 0.9$) considered in our study are aligned with the reference documentation, and 8.7% are overlapping. While the remaining 11.2% of assertions are not explicitly covered in the documen-

Table 6.9: The percentage of fully aligned, partially aligned, misaligned and irrelevant assertions

Case Study	Aligned	Overlapping	Unrelated		
			Low-risk	ODD Violation	Neither
NTSS	95%	0%	5%	0%	0%
AP–DHB	95%	0%	0%	5%	0%
AP–TWN (R1)	60%	15%	25%	0%	0%
AP–TWN (R2)	80%	15%	0%	5%	0%
AP–TWN (R3)	100%	0%	0%	0%	0%
AP–TWN (R4)	80%	5%	5%	10%	0%
AP–SNG	65%	15%	20%	0%	0%
DAVE2	65%	20%	15%	0%	0%
Average	80%	8.7%	8.7%	2.5%	0%

tation, i.e., labelled as unrelated, they still describe plausible low-risk situations or ODD-violation scenarios, according to our annotators. No assertions in our study were found to contradict the information in the reference documentation.

► Usefulness of the inferred assertions: We identify four ways in which the assertions inferred by our approach are useful to practitioners: *First*, assertions generated by GenTV provide SUT-specific quantified thresholds that are missing from the reference documentation, which typically only states qualitative ODD limits or high-level assumptions and leaves the choice of concrete thresholds to engineers. For example, for NTSS, the assertion in Table 6.8 identifies an upper bound (threshold) on aggregate traffic that can flow through high-priority classes without compromising QoS, a value that is not specified in the reference documentation on QoS and priority-based flow management [HJTM18, CAK25, ITU25]. Engineers of this system often have to select thresholds through limited ad-hoc observations rather than systematically collected data, which can yield inaccurate values that do not generalize beyond the scenarios engineers tried. Instead,

GenTV infers thresholds from a broader set of system executions, making them more robust and more likely to generalize than those chosen through ad-hoc observations. These thresholds can then guide system configuration and service-level agreements, for example, in the case of NTSS, by assigning flows to priority classes so that the inferred limits are not exceeded.

Second, assertions rated as unrelated capture plausible low-risk scenarios not covered explicitly in the reference documentation, where the system trivially behaves correctly. For example, for the AP-SNG system, GenTV identifies a safe or low-risk situation in which the ego vehicle is placed in the centre of a lane, starts at a very low speed, and drives on a straight road. Under these conditions, the vehicle keeps its lane. This scenario is not described as a specific safe case in the reference documentation, but GenTV classifies it as low risk based on the observed executions used to infer the assertions. For these low-risk situations, invoking the simulator provides limited insights.

Third, the inferred assertions can be integrated as runtime monitors – similar to self-oracles [SWCT20] – to check whether the system exceeds its ODD and to alert the human operator upon ODD violation. For example, for the DAVE2 system, GenTV infers that when the turning angle of the road exceeds 15 degrees, the ego vehicle fails to keep its lane. This assertion can be deployed as a runtime guard that detects entry into such road segments and issues a warning to the driver to take over or disengage the automated component.

Fourth, our approach generates assertions in a formal, machine-analyzable notation, enabling their direct use in automation. This reduces the effort and errors in extracting these assertions from natural language.

6.5.6 Threats to Validity

Internal Validity. To ensure a fair comparison among the different approaches, we use identical training sets and identical test sets across all experiments. We subjected DT and DR to systematic hyper-parameter tuning via Bayesian optimization [SLA12], and configured GP based on the best-practice recommendations from prior studies [LNS24, GMN+21, LP06]. To enable DT and DR to generate assertions with the same expressive power as those generated by GP, we performed feature engineering for DT and DR. This ensures that the conditions they learn for our case-study systems are comparable in structure and expressive power to those generated by our GP grammar. Consistent with prior research on identifying flaky tests [ANN24, KPT23, KPT24], we re-executed each test ten times to distinguish flaky ones. The test sets used to assess test-validator accuracy in RQ2 and RQ3 contain no flaky tests for systems with flakiness rates below 50%. Furthermore, for other systems, we minimized the presence of flaky tests in the test sets by including only those tests that achieved at least 80% consistent verdicts across re-executions. This approach to building test sets is not biased towards any specific test-validator generation technique. Further, since the accuracy of these techniques is assessed using the same test sets, we have not favoured any technique in the experiments for RQ3.

In RQ4, we use stratified sampling to select assertions for each case study. We chose a sample size of 20 because the experiments in RQ2 and RQ3 show that our test validators produce about 20 assertions on average in a single run. For RQ4, we require the logical assertions to be translated into natural language. Because accurate translation requires detailed knowledge of the domain concepts and system requirements – for example, in AP–DHB the pitchwheel signal refers to the angle of the aircraft's nose, and in NTSS, classes 5, 6, and 7 denote high-priority differentiated-services (DiffServ) classes – the translations were performed by the authors rather than automated tools or third-party annotators, whose lack of domain knowledge about our case studies would limit their ability to produce accurate translations. To mitigate this threat, the lead author produced the translations, and the other authors independently reviewed them to identify any changes in meaning and to maintain consistent terminology within each domain. We provide the complete set of translations in the supplementary material [git25].

Conclusion Validity. Since running multiple statistical tests can increase the risk of Type I error inflation [AB11], we apply the Benjamini-Hochberg (BH) procedure [BH95] to control the false discovery rate. In RQ2 and RQ3, all statistical significance tests were reported using BH-adjusted p-values.

External Validity. Our experiments are based on five different CPS and network systems. The aircraft autopilot system that we used in our evaluation is from the Lockheed Martin benchmark and has been previously used in the literature on testing CPS models [GPMS21, NGM+19]. Our NTSS case study is one of the few examples of industrial network systems in the literature [JNSS23, JCNS24]. The ADS systems we used in our evaluation are based on BeamNG, a leading open-source simulator widely referenced for

virtual and hybrid testing in ADS research, and used by the software testing community for benchmarking and competitions [bea25, sbf25]. In addition, one of our ADS relies on DAVE2 which is a DNN that has been successfully employed in real-world road testing conducted by NVIDIA [dav23]. Further experiments with a broader range of CPS would strengthen generalizability.

Our signal encoding assumes a piecewise constant interpolation function. Our evaluation of major CPS benchmarks and case studies from the literature indicates that this assumption is commonly made [KFA+24,loc25,cru25,clu25,gui25,dcm25]. While this choice is well suited to our case study domains, applying our encoding to systems in other domains may require adapting the interpolation strategy to better match domain-specific signal characteristics.

Limitation. GenTV is data-driven and the assertions it generates are based on empirical data rather than formal correctness guarantees. As a result, although in RQ4 we observed that GenTV did not produce assertions inconsistent with the reference documentation, we cannot guarantee that GenTV never generates such inconsistent assertions. To our knowledge, no formal techniques currently exist for deriving human-readable assertions that characterize input validity for complex CPS, such as those in our case studies. As discussed in Section 6.1, our work is related to prior data-driven approaches for determining the validity of test inputs for DL systems [RT23, GAT+25, DDS21] and for building self-oracles for ADS [SWCT20]. While these approaches rely on the probability that an input is out of distribution to identify invalid inputs or ODD violations, GenTV uses high-confidence assertions for similar purposes. In contrast to the prior work, the assertions generated by GenTV are human-readable. This enables domain experts to inspect them,

compare them against reference documentation using the procedure described in RQ4, and identify situations where the inferred assertions may be inconsistent with the documented requirements.

6.6 Summary

In this chapter, we introduced assertion-based test validators and evaluated their accuracy, robustness, and alignment with reference documentation. These validators reduce dependence on simulator executions by identifying conditions that characterize violations of ODD limits, unmet preconditions, and safe, low-risk scenarios, so executions are avoided for test inputs that fall in such conditions. Our construction process, based on GP and ML, yields validators that issue consistent verdicts and provide human-understandable explanations. We use spectrum-based fault localization formulas (Tarantula, Ochiai, Naish) as fitness functions in GP to evolve accurate and effective assertions. A formal analysis shows that assertion-based test validators have sufficient expressive power to capture a wide range of CPS properties. Our empirical evaluation across diverse domains shows that these validators deliver accurate and robust predictions even in the presence of test flakiness and that they align with technical standards and empirical studies. In particular, test validators generated by GP with Ochiai achieve high accuracy and robustness, with only a 4% variation in accuracy when inferred from training sets that include flaky tests. On average, 88.7% of the assertions inferred by our approach fully or partially align with requirement precondition violations, ODD-limit violations, and nominal safe conditions extracted from technical standards and empirical results.

Chapter 7

Related Work

In this chapter, we provide a structured review of the most relevant literature to contextualize and position the contributions of this thesis. Section 7.1 reviews robustness testing techniques and situates our approach in Chapter 5 (ENRICH) within this literature. Section 7.2 examines applications of ML in automated testing, focusing on its role in fuzzing, search-based testing, and the inference of failure-inducing rules. Section 7.3 reviews research on test input validity for DL-based systems, API testing, and assumption generation, and contrasts these approaches with our assertion-based test validators for CPS in Chapter 6 (GenTV). Section 7.4 outlines automated test-oracle inference techniques and discusses their relevance to assertion-based test validators. Section 7.5 considers spectrum-based fault localization (SBFL) ranking functions and their role in fault detection. Finally, Section 7.6 reviews studies on flaky tests, their causes, and mitigation techniques. Section 7.7 summarizes this chapter.

7.1 Robustness Testing

System robustness is considered an important engineering principle and has different implications on different development artifacts [SF13]. For example, to ensure robustness, system requirements should account for behaviours that allow a system to leave each of its failure states [SF10, JLHM90]; system design and implementation should include extensive error-handling [DS05, Iss92]; and, the development process should have a mechanism to predict and prevent robustness issues [AABB+03, LOV10]. Our approach, ENRICH, in Chapter 5 falls under the umbrella of robustness testing [PCL21, Koo98], which aims to determine whether a system functions properly in the presence of erroneous inputs or stressful environmental circumstances [SF13]. A common technique for robustness testing is fault injection. Fault injection for robustness testing spans both software (e.g., code mutation [LMX05]) and hardware (e.g., electromagnetic interferences [HHS+11] and power-supply disturbances [LL08]). For instance, Li et al. [LCH+94] evaluate the robustness of a telecommunication system by injecting software faults into the service manager; and, Barbosa et al. [BSDM07] employ fault injection to evaluate the robustness of third-party components at the interface level. For finding non-robust behaviours, instead of using fault injection, ENRICH relies on a robustness measure inspired by the search-based testing literature [HM09, LNB+17]. In addition to detecting individual cases of non-robustness, ENRICH identifies conditions under which a system exhibits non-robust behaviours.

The closest work to ENRICH is S-TaLiRo [ALFS11, NSF+10], which is a robustness testing tool for cyber-physical systems specified in Simulink. Through globally minimizing a robustness measure, S-TaLiRo generates counterexamples to a Simulink model's

temporal-logic requirements. The robustness measure employed by S-TaLiRo is the degree of perturbation that a Simulink model can withstand without changing the truth value of its specifications (expressed in temporal logic) [FP09, FP08, ALFS11, NSF+10]. In addition, a model satisfies (resp. dissatisfies) a specification *robustly* if its robustness measure is above (resp. below) zero [FP09, ALFS11, FP08, NSF+10]. While we adopt the general concept of robustness measure from S-TaLiRo, our work is different in three main ways:
(1) Our robustness measure is inspired by fitness computation in the search-based software testing (SBST) literature and differs from the temporal-logic robustness metric used by S-TaLiRo.
(2) S-TaLiRo focuses exclusively on falsification of Simulink models, i.e., identification of requirement failures for Simulink models. In contrast, ENRICH is applied to network traffic-shaping systems.
(3) Similar to most testing tools, S-TaLiRo generates individual test cases, whereas our approach is able to find ranges on input variables for non-robust behaviours.

At a conceptual level, our robustness measure bears some similarity to the notion of robustness in adversarial image classification [Wei20]. An image classification model behaves non-robustly if adding a small amount of noise (which is not recognizable by humans) changes the classification outcome for an image. Our work nonetheless differs in nature from adversarial image classification, since our inputs are streams of network packets and not images.

7.2 Applications of ML in Automated Testing

ML has been widely used to enhance the effectiveness of fuzz testing [MFS90] and search-based testing (SBT) [HM09]. In fuzz testing, ML has been employed to improve, among other things, seed generation, test sampling, and mutation-operator selection [HKL+10, WJL+20]. In SBT, surrogate models developed based on ML have been used to effectively and efficiently test CI systems such as CPS controllers and simulators [MNB17, AWM+17, HAK21, MNBP20], and autonomous-driving systems [BANBS16, BSH17, HKA22]. Surrogate modelling approaches relying on non-interpretable machine learning techniques include neural networks [BL88, BANBS16] and regression approaches such as polynomial regression [Sti74, MNB17], while those relying on interpretable techniques include decision trees and decision rules [YBOB22, ZWPL22]. These approaches demonstrate that using ML can improve the ability and the efficiency of testing in revealing faults. The ultimate goal of these approaches is to generate specific test cases. As such, these approaches are evaluated based on the number of failure-revealing tests and the severity of the failures, as determined by the fitness-function values.

Recent studies suggest that the focus of SBT should shift from generating a limited number of specific test cases to learning models that can explain system failures [FY20]. These models can then be employed for generating multiple test cases with specific properties. Motivated by these observations, our goal in Chapter 4 is to learn failure models and focus on improving the accuracy of these models for identifying spurious failures, rather than maximizing the number and severity of failure-revealing tests, which may not accu-

rately reflect the context where many tests fail due to spurious reasons. Below we review approaches for generating failure models and failure-inducing rules.

Grammar-based test generation [Han70] has been shown to be effective for avoiding spurious failures in fuzz testing. More recently, grammars and probabilistic variations of grammars have been used to infer abstract failure-inducing rules [KPAB22]. These approaches typically start with one or more known examples. They then generate additional tests iteratively, and infer assertions or grammars that explain the underlying causes of these failures. These rules can assist with the diagnosis of system failures, serve as accurate and high-level test oracles, and enable programmers to validate their fixes and prevent overfitting [KPAB22, GKH+20, KHSZ20]. Our work in Chapter 4 takes inspiration from the research on inferring failure-inducing rules, but differs from the existing work on this topic in important ways. First, we focus on systems with numeric inputs, whereas existing research primarily deals with string-based inputs governed by a grammar. Second, instead of relying on an input grammar to generate tests, we investigate various test-generation heuristics that are guided by quantitative fitness functions drawn from system requirements. An exception is the work of Böhme et al. [BGP20], which infers program patches for numeric systems without the need for input grammars. However, this approach relies on the availability of a human oracle to validate the verdicts of individual test inputs. In our context, this would be expensive and likely infeasible. Our work further differs from the above in that our goal is to identify rules for spurious failures rather than generating program patches.

The closest work to our approach in Chapter 4 is the Alhazen framework [KHSZ20], which we compared with in RQ3 in Section 4.4.4. In addition to the discussion and empir-

ical comparison in RQ3 in Section 4.4.4, we note that our approach differs from Alhazen in its input-feature engineering for failure models. We derive the input features for decision-rule models from domain-knowledge heuristics, whereas Alhazen derives the input features dynamically from its input grammar. While Alhazen automates input-feature engineering, by incorporating domain knowledge into the design of input features, our approach provides the flexibility to derive rules that more closely match expert intuition.

7.3 Test Input Validity

Traditionally, assessing the validity of test inputs has focused on identifying the preconditions and environmental assumptions of a system – i.e., the expectations a system makes about its operational context [GPB02]. Giannakopoulou et al. [GPB02] propose techniques for automatically generating assumptions that support compositional verification of software components. Cobleigh et al. [CGP03] extend this direction with learning-based methods that infer environment assumptions from counterexamples, enabling incremental assume–guarantee reasoning. In such settings, inputs that violate the learned or specified assumptions are treated as outside the environment model and are disregarded for property checking, which implicitly enforces a notion of test-input validity.

Recently, test input validity has been studied for DL systems. Several studies point out that many automatically generated inputs fall far outside the training data distribution and therefore should be considered invalid for performance assessment [RT23,DDS21,SWCT20]. These approaches primarily focus on DL systems whose test inputs are images, and propose to construct test validators using ML algorithms – e.g., variational autoencoders [DDS21,

SWCT20, RT23], vision systems metrics [HMC+22], or a combination of both [GAT+25]. Specifically, distribution-aware testing uses generative models to sample test inputs that remain close to the empirical input distribution and introduces validity checks to filter out unrealistic samples [DDS21]. Riccio and Tonella [RT23] empirically study when test input generators for DL produce invalid inputs, compare automated validators with human judgments, and show that invalid synthetic images are a major threat to the reliability of testing results.

Test-input validity for non-ML systems such as RESTful APIs includes checking that generated requests satisfy the syntactic and semantic constraints specified for the RESTful API under test. Mirabella et al. [MMLS+21] propose a DL approach that predicts whether a candidate API request is valid before invoking the service, using previous requests and responses to learn implicit inter-parameter dependencies and constraints that are not captured by the API specification. Such predictors filter out inputs likely to violate these constraints, thereby reducing the number of invalid requests produced by random or search-based test generators. Similarly, studies of online testing for web services report that a large fraction of randomly generated API calls are invalid and motivate the need for automated input validators [MLSRC22].

Test validators developed by our approach (GenTV) in Chapter 6 differ from prior work on input validity for assumption generation, DL models, and API testing in three main ways. *First*, DL-based approaches target vision systems whose inputs are images, API-oriented work operates over structured request parameters, and assumption-generation techniques reason over symbolic events or propositions, whereas our validators are designed for CPS whose inputs are continuous, high-dimensional numeric signals obtained from

simulators. *Second*, DL-based validators typically represent validity through latent features in learned models, and API validators often act as non-interpretable classifiers over request fields, while our validators are expressed as logical, human-readable assertions over input signals. *Third*, the purpose of DL test validators is to recognize inputs that the model has not been exposed to during training and to reject out-of-distribution or unrealistic samples, and API input validators aim to filter out requests that violate protocol or inter-parameter constraints. In contrast, our goal in Chapter 6 is to learn conditions that lead to violations of environmental assumptions, conditions characterizing safe or low-risk scenarios, or conditions that describe situations outside the SUT's ODD limits, and to use these conditions as test input validators for predicting pass or fail verdicts with high certainty.

7.4 Test Oracle

A test oracle is a mechanism that determines a pass or fail verdict for a given test input to a system. Traditionally, a human can serve as the oracle by defining the expected behaviour and manually judging outcomes, or by crafting explicit test oracles, a process that is often manual, laborious, and error-prone. Recent research aims to reduce the oracle cost problem by developing automated test oracles that generate verdicts without human intervention or system execution [HMSY13]. Dynamic analysis techniques have been proposed to infer such oracles [JCHT16, BHM+15, TJTP20]. For example, Ernst et al. [EPG+07] introduced Daikon, a dynamic analysis tool that observes program behaviour by running the software with various inputs and capturing values at different

points in the code. These values are analyzed to infer potential invariants, such as variable relationships and properties that hold across executions. Several studies use metamorphic testing to detect system failures by identifying violations of metamorphic relations (MRs) [HK18, KB13, KBBH16, NME19, ZZPL17, ATJ+24, TPJR18]. MRs predict expected outputs for different test inputs, and hence, can be used as test oracles [FG21]. In data-driven approaches to test oracle automation, test oracles are developed using ML models such as Adaptive Boosting [BNR+18, GBP23] or Neural Networks [GAH+18, GBP23, MNMK16, SKbI10, ASKS04, JWC+08, MZSK19, SBS11, YFZL06, ZWZ19].

Test validators developed by GenTV in Chapter 6 can be regarded as test oracles, since they decide whether the system passes or fails for a given test input. The key difference between automated test-oracle approaches and GenTV is that they infer test oracles based on a reference system rather than based on the SUT. Further, unlike most test oracles that require both inputs and outputs to determine verdicts, our test validators require only test inputs to issue verdicts.

7.5 SBFL Ranking Functions

Several studies use SBFL ranking functions to pinpoint faulty program statements [dSCK16]. There are more than 30 ranking functions studied in the SBFL literature [WGL+16, NLR11]. Among these, Tarantula, Ochiai and Naish have been studied more extensively compared to other ranking functions. Tarantula [JH05] is among the first ranking functions proposed for SBFL. Ochiai was originally used in molecular biology for measuring genetic similarity and was later adopted by SBFL for fault localization [AZVG07]. Naish has been intro-

duced more recently and has shown competitive performance compared to Ochiai in some studies [NLR11, MKKY14]. These functions have proven effective in correlating program statements (i.e., assertions in our work) with the pass and fail verdicts of a test suite.

Most SBFL ranking functions are designed manually [Yoo12]. GP has been employed to automatically generate SBFL ranking functions [Yoo12]. The ranking functions generated by GP are usually complex and involve various mathematical operators. This risks overfitting to specific SUT behaviours that only appear in the training sets. Consequently, using the GP-learned ranking functions to derive assertion-based test validators may lead to test validators that fail to generalize. In our work in Chapter 6, we show that test validators generated using Ochiai are more accurate than those generated using Tarantula or Naish, as well as those generated by DT and DR. Further, we are the first to explore the effectiveness of SBFL ranking functions as fitness functions of GP for generating accurate and robust test validators for CPS. SBFL ranking functions are particularly suitable as GP fitness functions for assertion generation, as they provide a smooth fitness landscape – an essential property for the effectiveness of GP [Luk13]. In contrast, fitness functions such as the one used by Gaaloul et al. [GMN+21] strictly prioritize soundness of candidate assertions, rewarding coverage only when no pass (resp. fail) tests satisfy a fail (resp. pass) candidate assertion. While this helps reduce incorrect predictions by assertions, it results in a non-smooth fitness landscape – small changes to a candidate assertion can cause abrupt shifts in fitness. As a result, fail (resp. pass) assertions that correctly capture a large portion of fail (resp. pass) tests but include a small number of pass (resp. fail) tests are penalized heavily and prematurely discarded. Heuristic search is less effective for fitness functions with a non-smooth landscape [Luk13].

7.6 Flaky Tests

Recent studies show that flaky tests are prevalent in both commercial and open-source software projects. For example, Google reported that nearly 16% of their 4.2 million test cases are flaky [Mic18], and the Microsoft Windows and Dynamics teams estimated a 5% rate of flaky test failures [HN15]. Prior research has explored the root causes of flaky tests and proposed techniques for addressing them across a range of systems, including open-source software [LHEM14], embedded systems [SOW+20], probabilistic programming systems [DSC+20], and ML frameworks [DSC+20, DSM21]. Common sources of flakiness include asynchronous waits, concurrency, test-order dependencies, and non-determinism in algorithmic behaviour. As shown in RQ1 (Section 6.5 in Chapter 6), our case-study systems – drawn from aerospace, networking, and autonomous driving domains – show varying levels of flakiness, often due to environmental variability, unpredictable hardware-software interactions, and underlying stochastic processes.

Several techniques have been developed to detect flaky tests, such as analyzing test execution order [LOS+19], identifying data dependencies among tests [GBZ18], and rerunning tests under different random seeds [DSC+20]. Our work in Chapter 6 differs from this line of research: rather than detecting or fixing flaky tests, we focus on evaluating the robustness of test validators learned from training sets that include flaky tests. Our robustness analysis helps determine whether flaky tests should be excluded from training sets or retained, depending on whether their inclusion significantly impacts the accuracy of the test validators. Our findings from RQ3 (Section 6.5.4 in Chapter 6) indicate that, when using GP with Ochiai, the average variation in prediction accuracy is around 4%. If

this variation is acceptable, flaky tests can remain in training sets, thereby reducing costs associated with detecting and removing them.

7.7 Summary

In this chapter, we review prior work in robustness testing, ML-based test generation, and test input validity, comparing them with the methods proposed in this thesis. Overall, we highlight that while existing research has laid important foundations in these areas, each strand exhibits limitations that our approaches address. Our approach in Chapter 5, ENRICH, extends robustness testing beyond fault injection by leveraging a search-inspired robustness measure to uncover input ranges where non-robust behaviours arise. Our work on learning failure models in Chapter 4 shifts the emphasis from isolated test-case generation to interpretable rule inference, enabling more accurate identification of spurious failures in systems with numeric inputs. In the domain of test input validity, our assertion-based test validator generation method in Chapter 6, GenTV, learns human-readable assertions that characterize realistic, valid operating conditions and filter out inputs that violate preconditions, exceed ODD limits, or yield vacuous low-risk scenarios. Finally, our analysis of the robustness of assertion-based test validators generated by GP demonstrates that meaningful test validators can be learned even in the presence of flaky tests, thereby reducing the cost of flakiness mitigation.

In the next chapter, we conclude this thesis by summarizing its key contributions and discussing future research opportunities.

Chapter 8

Conclusion

In this chapter, we summarize the contributions presented in this thesis and briefly discuss potential directions for future research.

8.1 Thesis Contributions

Cyber-physical systems (CPS) play a central role in safety-critical domains such as aerospace, autonomous driving, healthcare, and telecommunications, and their dependability directly impacts human safety and societal trust in automation. Traditional exhaustive testing is prohibitively expensive and infeasible due to the combinatorial explosion of possible system states and environmental conditions. Simulation-based testing, while offering a scalable and controlled environment, introduces challenges related to the effectiveness and reliability of the testing process. We define effectiveness through the prisms of capabil-

ity, interpretability, and efficiency, and reliability through trustworthy and realistic test outcomes.

In this thesis, we develop a set of methods that combine machine learning (ML) and search-based software engineering (SBSE) to strengthen the CPS testing workflow. Each method targets a critical challenge: efficient test generation, interpretable failure explanation, robustness analysis, and test input validity. Together, these contributions advance the state-of-the-art by reducing the cost of testing and improving the interpretability of test outcomes.

The contributions of this thesis are fourfold.

(1) Efficient test generation through surrogate modelling. We introduced a dynamic surrogate-assisted test generation approach that simultaneously leverages multiple surrogate models and dynamically selects predictions from the most accurate one (Chapter 4). This contribution reduces the dependence on costly simulator executions while maintaining predictive accuracy, thereby enabling broader exploration of CPS input spaces within a reasonable time and resource budget. This approach provides a foundation for scaling CPS testing to systems where each simulation is prohibitively expensive.

(2) Failure modelling and identification of spurious failures. Beyond detecting failures, we introduced approaches to construct interpretable failure models that capture the conditions leading to failures (Chapter 4). By combining SBSE with decision-rule learning, our approach distinguishes between genuine failures and spurious failures arising from invalid or unrealistic inputs. These models improve trust in testing results by filtering out misleading test outcomes.

(3) Learning and characterizing non-robust behaviours. We proposed ENRICH, a method for systematically identifying regions of the input space where small perturbations can cause the output to change from passing to failing and vice versa (Chapter 5). By characterizing non-robust behaviours, our method highlights the boundary conditions that require particular attention during system design and validation.

(4) Automated and interpretable test validator construction. We introduced GenTV an automated framework for constructing assertion-based test validators (Chapter 6). By employing GP and spectrum-based fault localization formulas, GenTV generates logical assertions over CPS inputs that characterize violations of requirement preconditions, inputs outside the system's operational design domain, and nominal low-risk scenarios in which requirements are trivially satisfied. The assertions are human-readable and formally expressive, making them suitable for safety-critical contexts where interpretability is crucial. Importantly, these validators maintain accuracy even under conditions of test flakiness.

Our approaches were validated through comprehensive case studies across domains of autonomous driving, networking, and industrial-scale Simulink models. Results demonstrated accuracy and efficiency improvements over baseline and state-of-the-art methods.

In addition to proposing approaches to address challenges related to the effectiveness and reliability of CPS testing, this thesis opens the door to a wide range of future research opportunities, which are outlined in the following section.

8.2 Opportunities for Future Research

The novel methods and findings of this thesis open several avenues for future research.

Enhancing explanation capabilities. In this thesis, we focused on using decision trees, decision rules and genetic programming to provide explanations for system behaviours. Future research can focus on integrating advanced techniques such as large language models (LLMs) with CPS testing workflows for automatically generating natural-language explanations of test verdicts or robustness boundaries. Such explanations can be natural language summaries that are tailored for different stakeholders (e.g., developers, safety engineers, or regulators). Furthermore, LLMs could be used to suggest potential root causes or mitigation strategies based on the identified failure conditions, creating an interactive diagnostic assistant.

In addition, future research can investigate the use of metamorphic testing (MT) to generate explanations for numeric-input CPS. MT defines metamorphic relations (MRs), properties describing how outputs should change under input transformations, which can help explain unexpected behaviours by showing where observed outputs violate expected transformation patterns.

Moreover, future work can study cross-simulator explanations, where the same CPS is exercised in multiple simulators (or simulator versions) and explanations are learned in a way that remains stable across these environments. This includes investigating how to align signals and verdict semantics across simulators, then extracting explanation factors that persist despite differences in fidelity, randomness, or physics engines. Such explana-

tions would be more likely to generalize beyond a single testbed, and can help distinguish simulator-specific artifacts from genuine system behaviours.

Further, our approaches in Chapters 4 and 6 focus on deriving interpretable explanations for the system behaviours as a one-time activity. Self-improving mechanisms can be adopted so that the explanations and test validators evolve over time based on feedback from new tests or system updates. This creates a self-improving testing loop where the validator can be refined based on its adherence to known system properties.

Advanced surrogate modelling. In this thesis, we demonstrated the effectiveness of using ensembles of surrogate models for test generation. A natural extension is to explore more sophisticated model architectures, particularly deep neural networks (DNNs), for capturing the complex, high-dimensional, and temporal behaviours of CPS. DNNs such as recurrent neural networks, convolutional neural networks and transformers could serve as powerful surrogates for simulators with long-term dependencies or multi-modal data.

Scalability and efficiency for complex CPS. Although our approaches demonstrated promising results on a range of case studies, the scalability of these methods to extremely large, heterogeneous, and highly interconnected CPS remains an open challenge. Future work could explore distributed and parallelized implementations of surrogate-assisted search, test-validator construction, and robustness characterization to handle industrial-scale systems. Research is needed to manage the computational complexity and communication overhead in such systems while maintaining the interpretability and reliability of the testing process.

Evaluation through user studies. An opportunity lies in evaluating the practical impact of our methods through longitudinal user studies and industrial deployments. While this thesis demonstrated effectiveness and efficiency through controlled case studies, user studies could assess how engineers and testers interact with the generated models, explanations, and test validators in real development environments. Such investigations could measure usability, interpretability, and decision-support capabilities in practice. Moreover, longitudinal studies in industrial settings would allow for an assessment of the long-term economic and organizational benefits of adopting ML- and SBSE-based testing methods. These evaluations would provide valuable evidence for bridging the gap between academic research and widespread industrial adoption.

User-interface support for practical adoption. Finally, a promising direction is to design an interactive user interface (UI) that integrates surrogate-assisted test generation, learned failure models, robustness characterization, and assertion-based test validators into a unified CPS testing dashboard. Such an interface could visualize explored and unexplored regions of the input space, highlight robustness boundaries, present human-readable validators alongside the requirements they operationalize, and allow engineers to inspect counterexamples and trace explanations back to specific signals and assumptions. By supporting feedback-driven refinement, the UI could enable users to confirm, adjust, or reject learned explanations and validators.

References

[AABB+03] Yamine Aït-Ameur, Gérard Bel, Frédéric Boniol, S Pairault, and Virginie Wiels. Robustness analysis of avionics embedded systems. *ACM SIGPLAN Notices*, 38(7):123–132, 2003.

[AB11] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 1–10, 2011.

[Adm09] Federal Aviation Administration. *Pilot's handbook of aeronautical knowledge*. Skyhorse Publishing Inc., 2009.

[ads25] 12 of the biggest failures in the driverless car industry in 2023. https://qz.com/a-timeline-of-all-the-ways-driverless-vehicles-failed-i-1851104020, (Accessed: December 2025).

[AFH+19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Network and Distributed System Security (NDSS) Symposium*, 2019.

[ALFS11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[alg25] Algorithms and boxplots. https://github.com/baharin/ENRICH/blob/main/Algorithms_and_Figures.pdf, (Accessed: December 2025).

[alh25a] Code to sota implementation for ntss case study. https://github.com/anonpaper23/testGenStrat/blob/main/Code/NTSS/SoTA.py, (Accessed: December 2025).

[alh25b] Code to sota implementation for simulink model case study. https://github.com/anonpaper23/testGenStrat/blob/main/Code/Simulink/Algorithms/decisiontreeSoTA.m, (Accessed: December 2025).

[alh25c] Replication package of alhazen framework. https://zenodo.org/records/3902142, (Accessed: December 2025).

[alh25d] Table 3 – parameter names, descriptions and values used by sota. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[ANBS18] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, pages 1016–1026. IEEE, 2018.

[ANN24] Mohammad Hossein Amini, Shervin Naseri, and Shiva Nejati. Evaluating the impact of flaky simulators on testing autonomous driving systems. *Empirical Software Engineering journal (EMSE)*, 29(2):1–30, 2024.

[ANSS24] Negin Ayoughi, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. Enhancing automata learning with statistical machine learning: A network security case study. In *Proceedings of the 27th International Conference on Model Driven Engineering Languages and Systems (MODELS 2024)*, pages 172–182, 2024.

[APN25] Rules obtained for each ci subject. https://github.com/anonpaper23/testGenStrat/blob/main/Evaluation%20Results/RQ4/APandNTSS_Rules.xlsx, (Accessed: December 2025).

[Arc13] Andrea Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[ARKA19] Muhammad Zubair Asghar, Fazal Rahman, Fazal Masud Kundi, and Shakeel Ahmad. Development of stock market trend prediction system using multiple regression. *Computational and mathematical organization theory*, 25(3):271–301, 2019.

[(AS09] Federal Aviation Administration (FAA)/Aviation Supplies & Academics (ASA). *Advanced Avionics Handbook*. FAA Handbooks Series. Aviation Supplies & Academics, Incorporated, 2009.

[ASKS04] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–6, 2004.

[ATJ+24] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. Genmorph: Automatically generating metamorphic relations via genetic programming. *IEEE Transactions on Software Engineering (TSE)*, 2024.

[aut25] Autopilot online benchmark. https://www.mathworks.com/matlabcentral/fileexchange/41490-autopilot-demo-for-arp4754a-do-178c-and-do-331?focused=6796756&tab=model, (Accessed: December 2025).

[AWM+17] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Search-based test case generation for cyber-physical systems.

In *Congress on Evolutionary Computation (CEC 2017)*, pages 688–697. IEEE, 2017.

[AWM+19] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.

[AZVG07] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[BALS20] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. Planning for untangling: Predicting the difficulty of merge conflicts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*, pages 801–811, 2020.

[BAN+21] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic–cross-replicating adas testing in two industry-grade automotive simulators. In *In proceeding of 14th International Conference on Software Testing, Verification and Validation (ICST 2021)*, pages 383–393. IEEE, 2021.

[BANBS16] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search

and neural networks. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE 2016)*, pages 63–74, 2016.

[BdTD+16] Mariusz Bojarski, David W. del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *ArXiv*, abs/1604.07316, 2016.

[bea25] Beamng.tech. https://beamng.tech, (Accessed: December 2025).

[BES+23] Rama Krishna Boyapati, Martha Morecock Eddy, Timothy Seitz, et al. Automated vehicles and adverse weather phase 3–final report. Technical report, United States. Department of Transportation. Federal Highway Administration, 2023.

[BFOS84] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. Classification and regression trees belmont. *CA: Wadsworth International Group*, 1984.

[BG11] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[BGK+23] Christian Birchler, Nicolas Ganz, Sajad Khatiri, Alessio Gambi, and Sebastiano Panichella. Cost-effective simulation-based test selection in self-driving cars software. *Science of Computer Programming*, 226:102926, 2023.

[BGP20] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. Human-in-the-loop automatic program repair. In *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST 2020)*, pages 274–285. IEEE, 2020.

[BH95] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995.

[BHM+15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[BKB+23] Christian Birchler, Sajad Khatiri, Bill Bosshard, Alessio Gambi, and Sebastiano Panichella. Machine learning-based test selection for simulation-based testing of self-driving cars software. *Empirical Software Engineering journal (EMSE)*, 28(3):71, 2023.

[BKD+23] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–30, 2023.

[BKR+25] Christian Birchler, Sajad Khatiri, Pooja Rani, Timo Kehrer, and Sebastiano Panichella. A roadmap for simulation-based testing of autonomous

cyber-physical systems: Challenges and future direction. *Transactions on Software Engineering and Methodology (TOSEM)*, 34(5):1–9, 2025.

[BL88] David S Broomhead and David Lowe. Radial basis functions, multivariable functional interpolation and adaptive networks. Technical report, Complex Systems, 2, 1988.

[BNPR20] Ravi Bhandari, Akshay Uttama Nambi, Venkata N Padmanabhan, and Bhaskaran Raman. Driving lane detection on smartphones using deep neural networks. *ACM Transactions on Sensor Networks (TOSN)*, 16(1):1–22, 2020.

[BNR+18] Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. A machine learning approach to generate test oracles. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering (SBES 2018)*, pages 142–151, 2018.

[BSAL17] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

[BSDM07] Ricardo Barbosa, Nuno Silva, João Duraes, and Henrique Madeira. Verification and validation of (real time) cots products using fault injection techniques. In *2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2007)*, pages 233–242. IEEE, 2007.

[BSH17] Halil Beglerovic, Michael Stolz, and Martin Horn. Testing of autonomous vehicles using surrogate models and stochastic optimization. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC 2017)*, pages 1–6. IEEE, 2017.

[BT24] Matteo Biagiola and Paolo Tonella. Testing of deep reinforcement learning agents with surrogate models. *Transactions on Software Engineering and Methodology (TOSEM)*, 33(3):1–33, 2024.

[cak] Common applications kept enhanced (cake) scheduler. https://github.com/dtaht/sch_cake. (Accessed: December 2025).

[CAK25] tc-cake. https://man7.org/linux/man-pages/man8/tc-cake.8.html, (Accessed: December 2025).

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[CGP03] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[Cha17] Devendra K Chaturvedi. *Modeling and simulation of systems using MATLAB® and Simulink®*. CRC press, 2017.

[clu25] Building a clutch lock-up model. https://www.mathworks.com/help/simulink/slref/building-a-clutch-lock-up-model.html, (Accessed: December 2025).

[Coh60] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[Coh95] William W Cohen. Fast effective rule induction. In *Machine learning proceedings 1995*, pages 115–123. Elsevier, 1995.

[cru25] Cruise control test generation. https://www.mathworks.com/help/sldv/ug/cruise-control-test-generation.html, (Accessed: December 2025).

[CWX24] Yuzhi Chen, Chen Wang, and Yuanchang Xie. Modeling the risk of single-vehicle run-off-road crashes on horizontal curves using connected vehicle data. *Analytic Methods in Accident Research*, 43:100333, 2024.

[DAB21] Arkadiy Dushatskiy, Tanja Alderliesten, and Peter AN Bosman. A novel surrogate-assisted evolutionary algorithm applied to partition-based ensemble learning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2021)*, pages 583–591, 2021.

[dat25] Raw datasets obtained from each algorithm for cps and ntss. https://github.com/anonpaper23/testGenStrat/tree/main/Data/Dataset, (Accessed: December 2025).

[dav23] End-to-End Deep Learning for Self-Driving Cars. https://developer.nvidia.com/blog/deep-learning-self-driving-cars/, 2023. (Accessed: December 2025).

[dcm25] Dc motor model simulink model. https://www.mathworks.com/matlabcentral/fileexchange/11587-dc-motor-model-simulink, (Accessed: December 2025).

[DDS21] Swaroopa Dola, Matthew B. Dwyer, and Mary Lou Soffa. Distribution-aware testing of neural networks using generative models. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 226–237. IEEE, 2021.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer, 2008.

[DMTBZTL16] Alan Díaz-Manríquez, Gregorio Toscano, Jose Hugo Barron-Zambrano, and Edgar Tello-Leal. A review of surrogate assisted multiobjective evolutionary algorithms. *Computational Intelligence and Neuroscience*, 2016(1):9420460, 2016.

[do-25] Navigating the do-178c certification process for airborne software. https://thecloudstrap.com/navigating-the-do-178c-certification-process/, (Accessed: December 2025).

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on Robot Learning (CoRL 2017)*, pages 1–16. PMLR, 2017.

[DS05] Hui Ding and Lui Sha. Dependency algebra: A tool for designing robust real-time systems. In *26th IEEE International Real-Time Systems Symposium (RTSS 2005)*, pages 11–pp. IEEE, 2005.

[DSC+20] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*, pages 211–224, 2020.

[dSCK16] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[DSM21] Saikat Dutta, August Shi, and Sasa Misailovic. Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, pages 603–614, 2021.

[EAD+19] Gidon Ernst, Paolo Arcaini, Alexandre Donzé, Georgios Fainekos, Logan Mathesen, Giulia Pedrielli, Shakiba Yaghoubi, Yoriyuki Yamagata, and

Zhenya Zhang. Arch-comp 2019 category report: Falsification. In Goran Frehse and Matthias Althoff, editors, *Proceedings of the 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 61 of *EPiC Series in Computing*, pages 129–140. EasyChair, 2019.

[enr25] Enrich – non-robustness analysis for traffic shaping. https://github.com/baharin/ENRICH, (Accessed: December 2025).

[EPG+07] 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.

[eva25a] Code to generate results of each research questions. https://github.com/anonpaper23/testGenStrat/tree/main/Evaluation, (Accessed: December 2025).

[eva25b] Results of each research question. https://github.com/anonpaper23/testGenStrat/tree/main/Evaluation%20Results, (Accessed: December 2025).

[FA12] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Transactions on Software Engineering (TSE)*, 39(2):276–291, 2012.

[FBBE16] Martina Friese, Thomas Bartz-Beielstein, and Michael Emmerich. Building ensembles of surrogates by optimal convex combination. *6th Inter-*

national Conference on Bioinspired Optimization Methods and Their Applications (BIOMA 2016), pages 131–143, 2016.

[FG21] Afonso Fontes and Gregory Gay. Using machine learning to generate test oracles: A systematic literature review. In *Proceedings of the 1st International Workshop on Test Oracles (TORACLE 2021)*, pages 1–10, 2021.

[fig25a] Figure 16 to figure 21 – precision and recall results obtained by varying time budget in rq2. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[fig25b] Figure 9 – comparing dataset sizes for dynamic sa algorithm and seven individual sa algorithms in rq1. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[FP08] Georgios E Fainekos and George J Pappas. A user guide for taliro. Technical report, Dept. of CIS, Univ. of Pennsylvania, 2008.

[FP09] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

[FY20] Robert Feldt and Shin Yoo. Flexible probabilistic modeling for search based test data generation. In *Proceedings of the 13th International*

Workshop on Search-Based Software Testing (SBST 2020), pages 537–540, 2020.

[Gaa21] Khouloud Gaaloul. *Verification of design models of cyber-physical systems specified in Simulink*. PhD thesis, 2021.

[GAH+18] Farshad Gholami, Niousha Attar, Hassan Haghighi, Mojtaba Vahidi Asl, Meysam Valueian, and Saina Mohamadyari. A classifier-based test oracle for embedded software. In *Real-Time and Embedded Systems and Technologies (RTEST 2018)*, pages 104–111. IEEE, 2018.

[GAT+25] Delaram Ghobari, Mohammad Hossein Amini, Dai Quoc Tran, Seunghee Park, Shiva Nejati, and Mehrdad Sabetzadeh. Test input validation for vision-based dl systems: An active learning approach. In *47th International Conference on Software Engineering (ICSE 2025)*, 2025. Available at https://arxiv.org/abs/2501.01606.

[GBP23] Charaka Geethal, Marcel Böhme, and Van-Thuan Pham. Human-in-the-loop automatic program repair. *Transactions on Software Engineering (TSE)*, 2023.

[GBZ18] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *Proceedings of 11th International Conference on Software Testing, Verification and Validation (ICST 2018)*, pages 1–11. IEEE, 2018.

[git25] Replication package for gentc. https://doi.org/10.5281/zenodo.17823049, (Accessed: December 2025).

[GKH+20] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2020)*, pages 237–248, 2020.

[GMH15] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of 37th IEEE International Conference on Software Engineering (ICSE 2015)*, volume 1, pages 789–800. IEEE, 2015.

[GMN+20] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C Briand, and David Wolfe. Mining assumptions for software components using machine learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pages 159–171, 2020.

[GMN+21] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache. Combining genetic programming and model checking to generate environment assumptions. *Transactions on Software Engineering (TSE)*, 48(9):3664–3685, 2021.

[GPB02] Dimitra Giannakopoulou, Corina S Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *International*

Conference on Automated Software Engineering (ASE 2002), pages 3–12. IEEE, 2002.

[GPMS21] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Automated formalization of structured natural language requirements. *Information and Software Technology*, 137:106590, 2021.

[gui21] Optimization studies in se (including search-based software engineering). https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=OptimizationStudies, 2021. (Accessed: December 2025).

[gui25] Design a guidance system in matlab and simulink. https://www.mathworks.com/help/simulink/slref/designing-a-guidance-system-in-matlab-and-simulink.html, (Accessed: December 2025).

[HAK21] Dmytro Humeniuk, Giuliano Antoniol, and Foutse Khomh. Data driven testing of cyber physical systems. In *Proceedings of 14th International Workshop on Search-Based Software Testing (SBST 2021)*, pages 16–19. IEEE, 2021.

[Han70] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[HHS+11] Yu-ichi Hayashi, Naofumi Homma, Takeshi Sugawara, Takaaki Mizuki, Takafumi Aoki, and Hideaki Sone. Non-invasive trigger-free fault injec-

tion method based on intentional electromagnetic interference. In *Non-Invasive Attack Testing Workshop (NIAT 2011)*, 2011.

[Him13] Andreas Himmler. Hardware-in-the-loop technology enabling flexible testing processes. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, page 816, 2013.

[HJTM18] Toke Høiland-Jørgensen, Dave Täht, and Jonathan Morton. Piece of cake: a comprehensive queue management solution for home gateways. In *International Symposium on Local and Metropolitan Area Networks (LANMAN 2018)*, pages 37–42. IEEE, 2018.

[HK18] Bonnie Hardin and Upulee Kanewala. Using semi-supervised learning for predicting metamorphic relations. In *Proceedings of the 3rd International Workshop on Metamorphic Testing (MET 2018)*, pages 14–17, 2018.

[HKA22] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. A search-based framework for automatic generation of testing environments for cyber-physical systems. *Information and Software Technology*, page 106936, 2022.

[HKL+10] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Proceedings of 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010)*, pages 182–191. IEEE, 2010.

[HLF22] Linxiong Hong, Huacong Li, and Jiangfeng Fu. A novel surrogate-model based active learning method for structural reliability analysis. *Computer Methods in Applied Mechanics and Engineering*, 394:114835, 2022.

[HM09] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Transactions on Software Engineering (TSE)*, 36(2):226–247, 2009.

[HMC+22] Boyue Caroline Hu, Lina Marsso, Krzysztof Czarnecki, Rick Salay, Huakun Shen, and Marsha Chechik. If a human can see it, so should your system: Reliability requirements for machine vision components. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, page 1145–1156, New York, NY, USA, 2022. Association for Computing Machinery.

[HMdSY12] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[HMSY13] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[HN15] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of 37th International Con-*

ference on Software Engineering (ICSE 2015), volume 2, pages 39–48. IEEE, 2015.

[HSNB21] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. Can offline testing of deep neural networks replace their online testing? a case study of automated driving systems. *Empirical Software Engineering journal (EMSE)*, 26(5):90, 2021.

[IBM25] What is gradient boosting? https://www.ibm.com/think/topics/gradient-boosting, (Accessed: December 2025).

[Iss92] Valérie Issarny. *An exception handling mechanism for parallel object-oriented programming: Towards the design of reusable and robust distributed software.* Inst. National de Recherche en Informatique et en Automatique, 1992.

[ITU25] Itu-t recommendation e.361, (Accessed: November 2025).

[JCHT16] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 247–258, 2016.

[JCNS24] Baharin A. Jodat, Abhishek Chandar, Shiva Nejati, and Mehrdad Sabetzadeh. Test generation strategies for building failure models and explaining spurious failures. *Transactions on Software Engineering and Methodology (TOSEM)*, 33(4):1–32, 2024.

[JGSN25] Baharin A. Jodat, Khouloud Gaaloul, Mehrdad Sabetzadeh, and Shiva Nejati. Automated test oracles for flaky cyber-physical system simulators: Approach and evaluation. *arXiv preprint arXiv:2508.20902*, 2025.

[JH05] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE 2005)*, pages 273–282, 2005.

[Jin05] Yaochu Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing*, 9(1):3–12, 2005.

[JLHM90] Matthew S Jaffe, Nancy G Leveson, Mats Heimdahl, and Bonnie Melhart. Software requirements analysis for real-time process-control systems. *UC Irvine: Donald Bren School of Information and Computer Sciences*, 1990.

[JNSS23] Baharin A. Jodat, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. Learning non-robustness using simulation-based testing: a network traffic-shaping case study. In *In proceedings of International Conference on Software Testing, Verification and Validation (ICST 2023)*, pages 386–397. IEEE, 2023.

[JS02] Yaochu Jin and Bernhard Sendhoff. Fitness approximation in evolutionary computation-a survey. In *Proceedings of the 4th Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1105–12, 2002.

[JWC+08] Hu Jin, Yi Wang, Nian-Wei Chen, Zhi-Jian Gou, and Shuo Wang. Artificial neural network for automatic test oracles generation. In *Proceedings of the International Conference on Computer Science and Software Engineering (CSSE 2008)*, volume 2, pages 727–730. IEEE, 2008.

[KB13] Upulee Kanewala and James M Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013)*, pages 1–10. IEEE, 2013.

[KBBH16] Upulee Kanewala, James M Bieman, and Asa Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software testing, verification and reliability*, 26(3):245–269, 2016.

[KDJ+16] James Kapinski, Jyotirmoy V Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *Control Systems Magazine*, 36(6):45–64, 2016.

[KFA+24] Tanmay Khandait, Federico Formica, Paolo Arcaini, Surdeep Chotaliya, Georgios Fainekos, Abdelrahman Hekal, Atanu Kundu, Ethan Lew, Michele Loreti, Claudio Menghi, Laura Nenzi, Giulia Pedrielli, Jarkko Peltomäki, Ivan Porres, Rajarshi Ray, Valentin Soloviev, Ennio Visconti,

Masaki Waga, and Zhenya Zhang. Arch-comp 2024 category report: Falsification. In Goran Frehse and Matthias Althoff, editors, *Proceedings of the 11th International Workshop on Applied Verification for Continuous and Hybrid Systems*, volume 103 of *EPiC Series in Computing*, pages 122–144. EasyChair, 2024.

[KHSZ20] Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pages 1228–1239, 2020.

[KLS21] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. Learning highly recursive input grammars. In *Proceedings of the 36th International Conference on Automated Software Engineering (ASE 2021)*, pages 456–467. IEEE, 2021.

[Koo98] Philip Koopman. Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security. In *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No. 98EX358)*, pages 103–131. IEEE, 1998.

[KPAB22] Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. Human-in-the-loop oracle learning for semantic bugs in string

processing programs. In *Proceedings of the 31st International Symposium on Software Testing and Analysis (ISSTA 2022)*, pages 215–226, 2022.

[KPT23] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In *Proceedings of the 16th International Conference on Software Testing, Verification and Validation (ICST 2023)*, pages 281–292. IEEE, 2023.

[KPT24] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based testing of unmanned aerial vehicles with aerialist. In *Proceedings of the 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 2024)*, pages 134–138, 2024.

[KTT17] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering journal (EMSE)*, 22(2):928–961, 2017.

[LCH+94] Tsanchi Li, Chi-Ming Chen, Bob Horgan, Ming Y Lai, and Steve Y Wang. A software fault insertion testing methodology for improving the robustness of telecommunications systems. In *Proceedings of the International Conference on Communications (ICC/SUPERCOMM 1994)*, pages 1767–1771. IEEE, 1994.

[LHEM14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 643–653, 2014.

[LL08] Carmine Landi and Mario Luiso. Performances assessment of electrical motors in presence of disturbances on power supply. In *Proceedings of the International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM 2008)*, pages 167–172. IEEE, 2008.

[LMN+24] Jia Li, Behrad Moeini, Shiva Nejati, Mehrdad Sabetzadeh, and Michael McCallen. A lean simulation framework for stress testing iot cloud systems. *IEEE Transactions on Software Engineering*, 2024.

[LMX05] Nik Looker, Malcolm Munro, and Jie Xu. A comparison of network level fault injection with code insertion. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 479–484. IEEE, 2005.

[LNB+16] Bing Liu, Shiva Nejati, Lionel Briand, Thomas Bruckmann, et al. Localizing multiple faults in simulink models. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, volume 1, pages 146–156. IEEE, 2016.

[LNB+17] Bing Liu, Shiva Nejati, Lionel C Briand, et al. Improving fault localization for simulink models using search-based testing and prediction models. In

Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), pages 359–370. IEEE, 2017.

[LNS24] Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Using genetic programming to build self-adaptivity into software-defined networks. *Transactions on Autonomous and Adaptive Systems (TAAS)*, 19(1):1–35, 2024.

[loc25] Lockheed martin. https://www.lockheedmartin.com, (Accessed: December 2025).

[log25] Logistic regression. http://faculty.cas.usf.edu/mbrannick/regression/Logistic.html, (Accessed: December 2025).

[LOS+19] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the 12th International Conference on Software Testing, Validation and Verification (ICST 2019)*, pages 312–322. IEEE, 2019.

[LOV10] Nuno Laranjeiro, Rui Oliveira, and Marco Vieira. Applying text classification algorithms in web services robustness testing. In *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS 2010)*, pages 255–264. IEEE, 2010.

[LP06] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary computation*, 14(3):309–344, 2006.

[LSN+22] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache. Estimating probabilistic safe wcet ranges of real-time systems at design stages. *Transactions on Software Engineering and Methodology (TOSEM)*, 2022.

[LSWH24] Guannan Lou, Donghwan Shin, Neil Walkinshaw, and Robert M Hierons. Autonomous driving system testing: Traffic density does matter. In *IFIP International Conference on Testing Software and Systems*, pages 315–331. Springer, 2024.

[Luk13] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. http://cs.gmu.edu/~sean/book/metaheuristics/.

[MAP19] CH Raga Madhuri, G Anuradha, and M Vani Pujitha. House price prediction using regression techniques: A comparative study. In *Proceedings of the International Conference on Smart Structures and Systems (ICSSS 2019)*, pages 1–5. IEEE, 2019.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[MFS90] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[Mic18] John Micco. Advances in continuous integration testing at google. 2018. https://research.google/pubs/advances-in-continuous-integration-testing-at-google/.

[MKKY14] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST 2014)*, pages 153–162. IEEE, 2014.

[MLSRC22] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of restful apis: Promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 408–420, 2022.

[MMLS+21] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. Deep learning-based prediction of test input validity for restful apis. In *2021 IEEE/ACM third international workshop on deep learning for testing and testing for deep learning (deepTest)*, pages 9–16. IEEE, 2021.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[MN10] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[MNB17] Reza Matinnejad, Shiva Nejati, and Lionel C. Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, (ESEC/FSE 2017)*, pages 938–943. ACM, 2017.

[MNBB14] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Brcukmann. Mil testing of highly configurable continuous controllers: Scalable search using surrogate models. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE 2014)*, page 163–174. ACM, 2014.

[MNBP20] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*, pages 372–384. IEEE, 2020.

[MNGB19] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 27–38, 2019.

[MNMK16] Wellington Makondo, Raghava Nallanthighal, Innocent Mapanga, and Prudence Kadebu. Exploratory test oracle using multi-layer perceptron

neural network. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI 2016)*, pages 1166–1171. IEEE, 2016.

[Mol20] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[MW47] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[MZSK19] Amin Karimi Monsefi, Behzad Zakeri, Sanaz Samsam, and Morteza Khashehchi. Performing software test oracle based on deep neural network with fuzzy inference system. In *Proceedings of the 2nd International Congress on High-Performance Computing and Big Data Analysis (TopHPC 2019)*, pages 406–417. Springer, 2019.

[Ng18] Andrew Ng. Machine learning yearning. 2018. http://www.mlyearning.org/.

[NGM+19] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 1015–1025, 2019.

[NHG21] Vuong Nguyen, Stefan Huber, and Alessio Gambi. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *Proceedings of the International Conference on Artificial Intelligence Testing (AITest 2021)*, pages 128–135. IEEE, 2021.

[NLR11] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):1–32, 2011.

[NME19] Aravind Nair, Karl Meinke, and Sigrid Eldh. Leveraging mutants for automatic prediction of metamorphic relations using machine learning. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2019)*, pages 1–6, 2019.

[NSF+10] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivancić, Aarti Gupta, and George J Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th International Conference on Hybrid Systems: Computation and Control (HSCC 2010)*, pages 211–220, 2010.

[NSS+23] Shiva Nejati, Lev Sorokin, Damir Safin, Federico Formica, Mohammad Mahdi Mahboob, and Claudio Menghi. Reflections on surrogate-assisted search-based testing: A taxonomy and two replication studies based on industrial ADAS and simulink models. *Information and Software Technology*, 163:107286, 2023.

[Ope25] OpenAI. Gpt-5 system card, Accessed: December 2025.

[PCL21] Justyna Petke, David Clark, and William B Langdon. Software robustness: a survey, a theory, and prospects. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, pages 1475–1478, 2021.

[PID] Tuning pid controller for self-driving cars. https://medium.com/@madhusudhan.d/tuning-pid-controller-for-self-driving-cars-3813f7f18eb0. (Accessed: December 2025).

[PKHM21] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

[PKKS19] Ripon Patgiri, Hemanth Katari, Ronit Kumar, and Dheeraj Sharma. Empirical study on malicious url detection using machine learning. In *Proceedings of the International Conference on Distributed Computing and Internet Technology (ICDCIT 2019)*, pages 380–388. Springer, 2019.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu.com, 2008. ISBN: 978-1-4092-0073-4.

[Pra21] SAE International Recommended Practice. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, April 2021.

[QHXJ22] Quan Quan, Zou Hao, Huang Xifeng, and Lei Jingchun. Research on water temperature prediction based on improved support vector regression. *Neural Computing and Applications*, 34(11):8501–8510, 2022.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[raq25] Raquel urtasun's tech company develops self-driving vehicle simulator. https://www.thestar.com/business/raquel-urtasun-s-tech-company-develops-self-driving-vehicle-simulator/article_4fc552f3-cbec-523c-ad3a-ec6aa93cdad7.html, (Accessed: December 2025).

[reg] Regression trees. https://www.solver.com/regression-trees. (Accessed: December 2025).

[req25] Cps and ntss requirements. https://github.com/anonpaper23/testGenStrat/blob/main/Benchmark/Formalization/CPS_and_NTSS_Formalization.pdf, (Accessed: December 2025).

[RT23] Vincenzo Riccio and Paolo Tonella. When and why test generators for deep learning produce invalid inputs: an empirical study. In *Proceedings*

of the 45th International Conference on Software Engineering (ICSE 2023), pages 1161–1173. IEEE, 2023.

[sbf25] Github repository for cps testing tool competition. https://github.com/sbft-cps-tool-competition/cps-tool-competition, (Accessed: December 2025).

[SBS11] Om Prakash Sangwan, Pradeep Kumar Bhatia, and Yogesh Singh. Radial basis function neural network based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–5, 2011.

[SF10] Ali Shahrokni and Robert Feldt. Towards a framework for specifying software robustness requirements based on patterns. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2010)*, pages 79–84. Springer, 2010.

[SF13] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1–17, 2013.

[SKbI10] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Suhaimi bin Ibrahim. An automated oracle approach to test decision-making structures. In *Proceedings of 3rd International Conference on Computer Science and Information Technology (ICCSIT 2010)*, volume 5, pages 30–34. IEEE, 2010.

[SKDB22] Sanaz Sheikhi, Edward Kim, Parasara Sridhar Duggirala, and Stanley Bak. Coverage-guided fuzz testing for cyber-physical systems. In *Proceed-*

ings of the 13th International Conference on Cyber-Physical Systems (ICCPS 2022), pages 24–33. IEEE, 2022.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[SM96] Ravindra Singh and Naurang Singh Mangat. Stratified sampling. In *Elements of survey sampling*, pages 102–144. Springer, 1996.

[SMP+18] Alexander Schaap, Gordon Marks, Vera Pantelic, Mark Lawford, Gehan Selim, Alan Wassyng, and Lucian Patcas. Documenting simulink designs of embedded systems. In *Proceedings on the 21st International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS 2018)*, page 47–51. ACM, 2018.

[sou25] Source codes of algorithms for cps and ntss. https://github.com/anonpaper23/testGenStrat/tree/main/Code, (Accessed: December 2025).

[SOW+20] Per Erik Strandberg, Thomas J Ostrand, Elaine J Weyuker, Wasif Afzal, and Daniel Sundmark. Intermittently failing tests in the embedded systems domain. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*, pages 337–348, 2020.

[SSK21] Chinmay Vilas Samak, Tanmay Vilas Samak, and Sivanathan Kandhasamy. Control strategies for autonomous vehicles. In *Autonomous driving and advanced driver-assistance systems (ADAS)*, pages 37–86. CRC Press, 2021.

[sta25] Results of statistical analysis. https://github.com/anonpaper23/testGenStrat/blob/main/Evaluation%20Results/RQ2/RQ2StatisticalResults.xlsx, (Accessed: December 2025).

[Sti74] Stephen M Stigler. Gergonne's 1815 paper on the design and analysis of polynomial regression experiments. *Historia Mathematica*, 1(4):431–439, 1974.

[SWCT20] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 359–371. ACM, 2020.

[SWH16] Robert C Streijl, Stefan Winkler, and David S Hands. Mean opinion score (mos) revisited: methods and applications, limitations and alternatives. *Multimedia Systems*, 22(2):213–227, 2016.

[SWYS11] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *Proceedings on the International Conference on*

Wireless Communications and Signal Processing (WCSP 2011), pages 1–6. IEEE, 2011.

[tab25a] Table 15 to table 20 – average accuracy, recall and precision over all runs of algorithms by varying execution time budget in rq2. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25b] Table 21 to table 24 – full set of rules obtained for ntss, ap1, ap2 and ap3 in rq4. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25c] Table 5 – time budgets given to non-ci subjects in rq1. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25d] Table 6 – statistical tests for dataset size and percentage of incorrect labels over dataset size in rq1. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25e] Table 7 – time budget considered for ci subjects in rq2. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25f] Table 8 – maximum number of test executions for non-ci subjects in rq2. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[tab25g] Table 9 to table 14 – statistical tests for accuracy, recall and precision by varying execution time budget in rq2. https://github.com/anonpaper23/testGenStrat/blob/main/Supplementary_Material.pdf, (Accessed: December 2025).

[TFP+19] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. Requirements-driven test generation for autonomous vehicles with machine learning components. *Transactions on Intelligent Vehicles (T-IV)*, 5(2):265–280, 2019.

[THMY21] Hao Tong, Changwu Huang, Leandro L Minku, and Xin Yao. Surrogate models in evolutionary single-objective optimization: A new taxonomy and experimental study. *Information Sciences*, 562:414–437, 2021.

[TJTP20] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pages 1178–1189, 2020.

[TPJR18] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Pro-*

ceedings of the 40th International Conference on Software Engineering (ICSE 2018), pages 303–314, 2018.

[tra] What is traffic shaping and why do i need it? https://net2phone.ca/resources/blog/what-is-traffic-shaping-and-why-do-i-need-it. (Accessed: December 2025).

[VD00] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[Vee13] Christian B. Veenhuis. Structure-based constants in genetic programming. In Luís Correia, Luís Paulo Reis, and José Cascalho, editors, *Progress in Artificial Intelligence*, pages 126–137, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[WAWZ25] Yuhang Wang, Abdulaziz Alhuraish, Shuyi Wang, and Hao Zhou. Empirical performance evaluation of lane keeping assist on modern production vehicles. *arXiv preprint arXiv:2505.11534*, 2025.

[WAYZ25] Yuhang Wang, Abdulaziz Alhuraish, Shengming Yuan, and Hao Zhou. Openlka: An open dataset of lane keeping assist from recent car models under real-world driving conditions. *arXiv preprint arXiv:2505.09092*, 2025.

[WCWL19] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*, pages 724–735. IEEE, 2019.

[Wei20] Kuo-An Andy Wei. *Understanding non-robust features in image classification*. PhD thesis, Massachusetts Institute of Technology, 2020.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Amsterdam, 3 edition, 2011.

[WGL+16] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016.

[WJD17] Handing Wang, Yaochu Jin, and John Doherty. Committee-based active learning for surrogate-assisted particle swarm optimization of expensive problems. *Transactions on Cybernetics*, 47(9):2664–2677, 2017.

[WJL+20] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[xpl25] X-plane flight simulator. https://www.x-plane.com/, (Accessed: December 2025).

[XZLX21] Huanwei Xu, Xin Zhang, Hao Li, and Ge Xiang. An ensemble of adaptive surrogate models based on local error expectations. *Mathematical Problems in Engineering*, 2021, 2021.

[YBOB22] Jun Yuan, Brian Barr, Kyle Overton, and Enrico Bertini. Visual exploration of machine learning model behavior with hierarchical surrogate rule sets. *Transactions on Visualization and Computer Graphics (TVCG)*, 30(2):1470–1488, 2022.

[YFZL06] Mao Ye, Boqin Feng, Li Zhu, and Yao Lin. Automated test oracle based on neural networks. In *Proceedings of 5th International Conference on Cognitive Informatics*, volume 1, pages 517–522. IEEE, 2006.

[Yoo12] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE 2012)*, pages 244–258. Springer, 2012.

[YSYA19] Keisuke Yoneda, Naoki Suganuma, Ryo Yanase, and Mohammad Aldibaja. Automated driving recognition technologies for adverse weather conditions. *IATSS research*, 43(4):253–262, 2019.

[Zel17] Andreas Zeller. Search-based testing and system testing: a marriage in heaven. In *Proceedings of 10th International Workshop on Search-Based Software Testing (SBST 2017)*, pages 49–50. IEEE, 2017.

[ZWPL22] Xiubin Zhu, Dan Wang, Witold Pedrycz, and Zhiwu Li. Fuzzy rule-based local surrogate models for black-box model explanation. *Transactions on Fuzzy Systems*, 31(6):2056–2064, 2022.

[ZWZ19] Ran Zhang, Ya-wen Wang, and Ming-zhe Zhang. Automatic test oracle based on probabilistic neural networks. In *Proceedings of the Recent Developments in Intelligent Computing, Communication and Devices (ICCD 2017)*, pages 437–445. Springer, 2019.

[ZZPL17] Pengcheng Zhang, Xuewu Zhou, Patrizio Pelliccione, and Hareton Leung. Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network. *IEEE access*, 5:21791–21805, 2017.

APPENDICES

Appendix A

Equivalence between logic fragment $\mathcal{L}$ and grammar $\mathcal{G}$

We show equivalence by constructing bidirectional transformations between any formula in $\mathcal{L}$ and conditions expressed by the grammar $\mathcal{G}$ defined in Figure 6.4 in Chapter 6.

(1) Transforming a formula in $\mathcal{L}$ to a condition based on grammar $\mathcal{G}$. We show that any formula ϕ written in $\mathcal{L}$ can be expressed as a condition based on our grammar $\mathcal{G}$.

Proof. We proceed by structural induction on the structure of $\phi \in \mathcal{L}$.

Base Cases.

- **Base Case 1: $\phi = \mathbf{r} \sim \mathbf{0}$**

  Let $\phi = r \sim 0$ where $\sim \in \{<, \leq, >, \geq, =, \neq\}$. Based on the definition of $\mathcal{L}$ in Sec-

tion 6.4.3, we have $r \in \mathbb{R}$. Therefore, r maps to const in grammar $\mathcal{G}$. Thus, $r \sim 0$ corresponds to const ~ 0, which is a rel-term in grammar $\mathcal{G}$.

- **Base Case 2:** $\phi = \forall \mathbf{t} \in \langle \mathbf{n_1}, \mathbf{n_2} \rangle : \mathbf{u(t)} \sim \mathbf{0}$

  Let $\phi = \forall t \in \langle n_1, n_2 \rangle : u(t) \sim 0$, where u is a signal over the time domain $\mathbb{T} = [0, b]$, and $\langle n_1, n_2 \rangle$ is a subinterval of $\mathbb{T}$. Based on Section 6.4.2, u is encoded using n_u control points, i.e., $c_{u,0}, \ldots, c_{u,n_u-1}$ each placed at time instants $0, I, 2 \cdot I, \ldots, (n_u - 1) \cdot I$, where $I = \frac{b}{n_u - 1}$. Under the piecewise constant interpolation assumption, the value of $u(t)$ is constant within each interval $[j \cdot I, (j+1) \cdot I)$ and equal to $c_{u,j}$.

  If the interval $\langle n_1, n_2 \rangle$ is contained within a single time interval $[j \cdot I, (j+1) \cdot I)$, then $\forall t \in \langle n_1, n_2 \rangle : u(t)$ maps to $c_{u,j}$ which is control point cp in grammar $\mathcal{G}$. Thus, $\forall t \in \langle n_1, n_2 \rangle : u(t) \sim 0$ corresponds to cp ~ 0 which is a rel-term in grammar $\mathcal{G}$.

  If $\langle n_1, n_2 \rangle$ spans multiple control-point intervals (e.g., $[j \cdot I, (j+k) \cdot I)$ for some $k \geq 2$), then $\forall t \in \langle n_1, n_2 \rangle : u(t) \sim 0$ can be decomposed into conjunctions over each unit interval:

  $$\forall t \in [j \cdot I, (j+1) \cdot I) : u(t) \sim 0 \ \wedge \ldots \wedge \forall t \in [(j+k-1) \cdot I, (j+k) \cdot I) : u(t) \sim 0.$$

  Each subformula $\forall t \in [i \cdot I, (i+1) \cdot I) : u(t) \sim 0$ is semantically equivalent to $c_{u,i} \sim 0$. Hence, the entire formula corresponds to:

  $$c_{u,j} \sim 0 \ \wedge c_{u,j+1} \sim 0 \ \wedge \ldots \wedge c_{u,j+k-1} \sim 0$$

  which is an and-term over rel-terms in grammar $\mathcal{G}$.

- **Base Case 3:** $\phi = \forall \mathbf{t} \in \langle \mathbf{n_1}, \mathbf{n_2} \rangle : \rho \sim \mathbf{0}$

  Based on the definition of $\mathcal{L}$ in Section 6.4.3, $\rho ::= u(t) \mid r \mid \rho_1 + \rho_2 \mid \rho_1 - \rho_2 \mid \rho_1 \times \rho_2 \mid \rho_1 / \rho_2$. By the inductive structure of ρ, ρ_1 and ρ_2 are each either $u(t)$ or a constant r, or further composed of arithmetic terms. We note that based on Section 6.4.3, $\mathcal{L}$ does not allow nested quantifiers, hence, no additional quantifiers (i.e., $\forall t$) appear in ϕ.

  As in Base Cases 1 and 2, each occurrence of $u(t)$ within the interval $\langle n_1, n_2 \rangle$ can be replaced with the corresponding control point $c_{u,j}$. Further, each occurrence of r remains as is. Therefore, ρ_1 and ρ_2 can be rewritten into arithmetic expressions over control points and constants, i.e., valid exps in $\mathcal{G}$. Hence, the formula $\forall t \in \langle n_1, n_2 \rangle : \rho \sim 0$ corresponds to an expression like exp ~ 0 over control points and constants, which is a rel-term in grammar $\mathcal{G}$.

Inductive Cases.

- **Inductive Case 1:** $\phi = \phi_1 \wedge \phi_2$

  The inductive hypothesis is that ϕ_1 and ϕ_2 correspond to rel-terms or and-terms in grammar $\mathcal{G}$. Since we assume that no free time variables exist in ϕ_1 and ϕ_2 based on Section 6.4.3, ϕ_1 and ϕ_2 ultimately reduce to one of the base cases—either a simple relational expression over constants (Base Case 1), over control points (Base Case 2) or over arithmetic expressions (Base Case 3). Hence, $\phi_1 \wedge \phi_2$ is a conjunction of expressions derived from base cases and hence corresponds to an and-term in grammar $\mathcal{G}$.

- **Inductive Case 2:** $\phi = \phi_1 \vee \phi_2$

  Similar to Inductive Case 1, by the inductive hypothesis, both ϕ_1 and ϕ_2 correspond to rel-terms or and-terms in grammar $\mathcal{G}$. As there are no free time variables, all expressions in ϕ_1 and ϕ_2 are grounded in the base cases, making them structurally reducible to rel-terms or and-terms. Hence, $\phi_1 \vee \phi_2$ is a disjunction of expressions derived from base cases and hence corresponds to an or-term in grammar $\mathcal{G}$.

Therefore, every formula $\phi \in \mathcal{L}$ corresponds to a well-formed condition in grammar $\mathcal{G}$. □

(2) Transforming a condition in $\mathcal{G}$ to a formula in $\mathcal{L}$. The transformation of any condition in $\mathcal{G}$ into a formula in $\mathcal{L}$ can be established analogously, using structural induction on the grammar $\mathcal{G}$. Briefly,

- Each control point cp corresponds to a sub-interval of the time domain where the signal $u(t)$ is constant; thus, each control point maps to a formula $\forall t \in \langle n_1, n_2 \rangle : u(t)$ in $\mathcal{L}$ by the piecewise constant assumption. Further, each const corresponds to r in logic $\mathcal{L}$.

- Each rel-term corresponds directly to a formula of the form $\rho \sim 0$ in $\mathcal{L}$, where ρ is an arithmetic expression over signals and constants.

- Each and-term corresponds to conjunction $\phi_1 \wedge \phi_2$, and each or-term corresponds to disjunction $\psi_1 \vee \psi_2$ in $\mathcal{L}$.

Hence, the transformation from grammar $\mathcal{G}$ to logic $\mathcal{L}$ follows the same structural induction as in (1) and is omitted for brevity.

Conclusion. Based on (1) and (2), we have equivalence between logic fragment $\mathcal{L}$ and grammar $\mathcal{G}$, meaning that for every formula in $\mathcal{L}$ there exists a condition in $\mathcal{G}$, and for every condition in $\mathcal{G}$ there exists a formula in $\mathcal{L}$, with both transformations preserving the original semantics.

Appendix B

Supplementary results for RQ2 and RQ3 in Chapter 6

This section presents the statistical-test results for RQ2 and RQ3 in Sections 6.5.3 and 6.5.4 of Chapter 6. Detailed discussions of these tables are provided in Sections 6.5.3 and 6.5.4.

Table B.1: Statistical tests comparing the accuracy results of GP_O against those of GP_T, GP_N, DT, DR and ensemble. The p-values highlighted in blue represent cases where GP_O significantly outperforms the compared alternative. The significance level is 0.05.

θ	GP_O **vs** GP_T		GP_O **vs** GP_N		GP_O **vs DT**		GP_O **vs DR**		GP_O **vs Ensemble**	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
0.5	1.05E-25 $GP_O > GP_T$	0.86 (L)	0.03 $GP_O > GP_N$	0.57 (S)	1.59E-25 $GP_O > DT$	0.85 (L)	4.3E-45 $GP_O > DR$	0.98 (L)	7.83E-21 $GP_O > Ensemble$	0.82 (L)
0.55	3.59E-26 $GP_O > GP_T$	0.87 (L)	0.01 $GP_O > GP_N$	0.59 (S)	2.14E-25 $GP_O > DT$	0.85 (L)	2.56E-44 $GP_O > DR$	0.97 (L)	1.98E-19 $GP_O > Ensemble$	0.81 (L)
0.6	1.5E-21 $GP_O > GP_T$	0.84 (L)	0.04 $GP_O > GP_N$	0.57 (S)	3.33E-22 $GP_O > DT$	0.84 (L)	5.07E-41 $GP_N > DR$	0.98 (L)	7E-16 $GP_O > Ensemble$	0.79 (L)
0.65	9.03E-23 $GP_O > GP_T$	0.88 (L)	5.55E-08 $GP_O > GP_N$	0.71 (L)	7.74E-18 $GP_O > DT$	0.83 (L)	3.68E-36 $GP_O > DR$	0.98 (L)	9.48E-18 $GP_O > Ensemble$	0.83 (L)
0.7	1.19E-24 $GP_O > GP_T$	0.92 (L)	1.51E-07 $GP_O > GP_N$	0.73 (L)	3.37E-16 $GP_O > DT$	0.84 (L)	8.42E-32 $GP_O > DR$	0.98 (L)	1.6E-20 $GP_O > Ensemble$	0.88 (L)
0.75	1.24E-19 $GP_O > GP_T$	0.94 (L)	4.42E-11 $GP_O > GP_N$	0.85 (L)	1.09E-14 $GP_O > DT$	0.87 (L)	1.14E-22 $GP_O > DR$	0.97 (L)	1.51E-17 $GP_O > Ensemble$	0.91 (L)
0.8	2.9E-14 $GP_O > GP_T$	0.94 (L)	6.45E-06 $GP_O > GP_N$	0.78 (L)	3.33E-08 $GP_O > DT$	0.81 (L)	6.42E-17 $GP_O > DR$	0.97 (L)	2.33E-11 $GP_O > Ensemble$	0.88 (L)
0.85	3.61E-13 $GP_O > GP_T$	0.96 (L)	0.004 $GP_O > GP_N$	0.72 (L)	3.15E-08 $GP_O > DT$	0.84 (L)	2.47E-14 $GP_O > DR$	0.97 (L)	3.3E-10 $GP_O > Ensemble$	0.89 (L)
0.9	0.0004 $GP_O > GP_T$	0.85 (L)	0.97 $GP_O \approx GP_N$	0.50 (N)	0.81 $GP_O \approx DT$	0.52 (N)	0.0003 $GP_O > DR$	0.85 (L)	0.12 $GP_O \approx Ensemble$	0.65 (M)
0.95	0.001 $GP_O > GP_T$	0.89 (L)	0.67 $GP_O \approx GP_N$	0.57 (S)	0.84 $GP_O \approx DT$	0.47 (N)	0.0002 $GP_O > DR$	0.93 (L)	0.16 $GP_O \approx Ensemble$	0.66 (M)
1	0.001 $GP_O > GP_T$	0.91 (L)	0.74 $GP_O \approx GP_N$	0.57 (S)	0.28 $GP_O \approx DT$	0.36 (M)	0.001 $GP_O > DR$	0.91 (L)	0.18 $GP_O \approx Ensemble$	0.67 (M)

Table B.2: Statistical tests comparing the accuracy results of GP_O against those of GP_T, GP_N, DT, DR and ensemble for each study subject. The p-values highlighted in blue represent cases where GP_O significantly outperforms the compared alternative. The p-values highlighted in orange indicate cases where the compared alternative outperforms GP_O. The cells highlighted in yellow represent cases where DT is not applicable. The significance level is 0.05.

Study Subject	GP_O **vs** GP_T		GP_O **vs** GP_N		GP_O **vs DT**		GP_O **vs DR**		GP_O **vs Ensemble**	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
Router	1.9E-33 $GP_O > GP_T$	0.87 (L)	8.4E-05 $GP_O > GP_N$	0.62 (S)	4.55E-15 $DT > GP_O$	0.29 (L)	4.72E-51 $GP_O > DR$	0.91 (L)	6.58E-07 $GP_O > Ensemble$	0.65 (M)
AP–DHB	0.0001 $GP_O > GP_T$	0.68 (L)	0.16 $GP_N \approx GP_O$	0.43 (S)	1.64E-59 $GP_O > DT$	1 (L)	2.75E-26 $GP_O > DR$	1 (L)	1.93E-09 $GP_O > Ensemble$	0.77 (L)
AP–TWN (R1)	1.37E-11 $GP_O > GP_T$	1 (L)	6.69E-11 $GP_O > GP_N$	1 (L)	9.01E-12 $GP_O > DT$	1 (L)	4.42E-13 $GP_O > DR$	1 (L)	2.06E-11 $GP_O > Ensemble$	1 (L)
AP–TWN (R2)	0.002 $GP_O > GP_T$	0.60 (S)	0.17 $GP_N \approx GP_O$	0.54 (N)	1.83E-12 $GP_O > DT$	0.72 (L)	7.52E-66 $GP_O > DR$	1 (L)	0.01 $Ensemble > GP_O$	0.41 (S)
AP–TWN (R3)	9.1E-33 $GP_O > GP_T$	1 (L)	1.01E-22 $GP_O > GP_N$	0.91 (L)	3.66E-69 $GP_O > DT$	1 (L)	4.72E-51 $GP_O > DR$	1 (L)	3.48E-43 $GP_O > Ensemble$	1 (L)
AP–TWN (R4)	1.41E-33 $GP_O > GP_T$	1 (L)	2.03E-15 $GP_O > GP_N$	0.82 (L)	DT is not applicable		6.13E-21 $GP_O > DR$	1 (L)	8.76E-33 $GP_O > Ensemble$	1 (L)
AP–SNG	8.5E-46 $GP_O > GP_T$	1 (L)	2.48E-42 $GP_O > GP_N$	1 (L)	7.42E-68 $GP_O > DT$	1 (L)	1.17E-50 $GP_O > DR$	1 (L)	1.83E-58 $GP_O > Ensemble$	1 (L)
Dave2	7.11E-13 $GP_O > GP_T$	0.78 (L)	0.02 $GP_N > GP_O$	0.40 (S)	1.24E-46 $GP_O > DT$	1 (L)	1.23E-22 $GP_O > DR$	0.86 (L)	6.78E-09 $GP_O > Ensemble$	0.72 (L)

Table B.3: Statistical test results comparing (a) the rate of pass verdicts predicted as fail achieved with DR against those of GP_T, GP_O, GP_N, DT and ensemble (b) the rate of fail verdicts predicted as pass achieved with GP_O against those of GP_T, GP_N, DT, DR and ensemble. The p-values highlighted in blue represent cases where the method on the left significantly outperforms the compared alternative. The p-values highlighted in orange indicate cases where the compared alternative significantly outperforms the method on the left. The significance level is 0.05.

(a) Rate of pass verdicts predicted as fail

θ	DR vs GP_T		DR vs GP_O		DR vs GP_N		DR vs DT		DR vs Ensemble	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
0.5	2.85E-11 $DR > GP_T$	0.29 (L)	3.56E-15 $DR > GP_O$	0.24 (L)	6.28E-18 $DR > GP_N$	0.22 (L)	1.11E-10 $DR > DT$	0.30 (M)	1.32E-12 $DR > Ensemble$	0.28 (L)
0.55	3.72E-11 $DR > GP_T$	0.29 (L)	5.52E-15 $DR > GP_O$	0.24 (L)	3.83E-17 $DR > GP_N$	0.22 (L)	3E-10 $DR > DT$	0.30 (M)	1.1E-12 $DR > Ensemble$	0.28 (L)
0.6	9.75E-11 $DR > GP_T$	0.30 (M)	3.78E-14 $DR > GP_O$	0.24 (L)	3.83E-17 $DR > GP_N$	0.22 (L)	3E-10 $DR > DT$	0.30 (M)	1.09E-12 $DR > Ensemble$	0.28 (L)
0.65	3.24E-10 $DR > GP_T$	0.30 (M)	2.2E-13 $DR > GP_O$	0.23 (L)	2.31E-08 $DR > GP_N$	0.31 (M)	5.28E-06 $DR > DT$	0.35 (M)	2.09E-07 $DR > Ensemble$	0.34 (M)
0.7	3.94E-08 $DR > GP_T$	0.32 (M)	5.93E-22 $DR > GP_O$	0.11 (L)	3.68E-20 $DR > GP_N$	0.16 (L)	5.01E-06 $DR > DT$	0.35 (M)	1.83E-07 $DR > Ensemble$	0.34 (M)
0.75	1.88E-05 $DR > GP_T$	0.36 (M)	2.76E-12 $DR > GP_O$	0.17 (L)	2.52E-15 $DR > GP_N$	0.19 (L)	1.99E-05 $DR > DT$	0.36 (M)	0.0007 $DR > Ensemble$	0.39 (S)
0.8	0.0001 $DR > GP_T$	0.37 (S)	1.64E-13 $DR > GP_O$	0.10 (L)	3.82E-13 $DR > GP_N$	0.20 (L)	1.9E-05 $DR > DT$	0.36 (M)	6.5E-05 $DR > Ensemble$	0.37 (S)
0.85	0.0003 $DR > GP_T$	0.37 (S)	9.42E-14 $DR > GP_O$	0.06 (L)	3.19E-15 $DR > GP_N$	0.05 (L)	1.23E-05 $DR > DT$	0.35 (M)	7.44E-05 $DR > Ensemble$	0.37 (S)
0.9	4.3E-05 $DR > GP_T$	0.35 (M)	0.002 $DR > GP_O$	0.23 (L)	8.94E-07 $DR > GP_N$	0.11 (L)	3.26E-06 $DR > DT$	0.34 (M)	5.12E-05 $DR > Ensemble$	0.36 (M)
0.95	0.0006 $DR > GP_T$	0.37 (S)	0.01 $DR > GP_O$	0.23 (L)	6.06E-06 $DR > GP_N$	0.07 (L)	0.0003 $DR > DT$	0.37 (S)	0.003 $DR > Ensemble$	0.39 (S)
1	2.82E-05 $DR > GP_T$	0.33 (M)	0.03 $DR > GP_O$	0.26 (L)	3.69E-05 $DR > GP_N$	0.09 (L)	2.13E-05 $DR > DT$	0.33 (M)	0.001 $DR > Ensemble$	0.38 (S)

(b) Rate of fail verdicts predicted as pass

θ	GP_O vs GP_T		GP_O vs GP_N		GP_O vs DT		GP_O vs DR		GP_O vs Ensemble	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
0.5	0.03 $GP_O > GP_T$	0.43 (S)	0.07 $GP_O \approx GP_N$	0.43 (S)	0.02 $GP_O > DT$	0.42 (S)	0.0006 $GP_O > DR$	0.38 (S)	0.08 $GP_O \approx Ensemble$	0.44 (N)
0.55	0.61 $GP_O \approx GP_T$	0.48 (N)	0.06 $GP_O \approx GP_N$	0.43 (S)	0.72 $GP_O \approx DT$	0.51 (N)	0.20 $GP_O \approx DR$	0.45 (N)	0.88 $GP_O \approx Ensemble$	0.50 (N)
0.6	0.03 $GP_O > GP_T$	0.43 (S)	0.001 $GP_O > GP_N$	0.39 (S)	0.03 $GP_O > DT$	0.43 (S)	0.0001 $GP_O > DR$	0.36 (M)	0.06 $GP_O \approx Ensemble$	0.43 (S)
0.65	0.02 $GP_O > GP_T$	0.42 (S)	0.03 $GP_O \approx GP_N$	0.42 (S)	0.0005 $GP_O > DT$	0.37 (S)	0.0002 $GP_O > DR$	0.36 (M)	0.01 $GP_O > Ensemble$	0.41 (S)
0.7	3.47E-06 $GP_O > GP_T$	0.34 (M)	0.008 $GP_O > GP_N$	0.42 (S)	9.12E-15 $GP_O > DT$	0.19 (L)	1.91E-13 $GP_O > DR$	0.21 (L)	8.52E-10 $GP_O > Ensemble$	0.27 (L)
0.75	0.001 $GP_O > GP_T$	0.38 (S)	0.27 $GP_O \approx GP_N$	0.52 (N)	7.6E-11 $GP_O > DT$	0.19 (L)	7.67E-08 $GP_O > DR$	0.25 (L)	6.19E-07 $GP_O > Ensemble$	0.28 (L)
0.8	0.32 $GP_O \approx GP_T$	0.46 (N)	0.05 $GP_O \approx GP_N$	0.54 (N)	7.79E-07 $GP_O > DT$	0.22 (L)	0.19 $GP_O \approx DR$	0.44 (N)	0.0005 $GP_O > Ensemble$	0.32 (M)
0.85	0.89 $GP_O \approx GP_T$	0.50 (N)	0.63 $GP_O \approx GP_N$	0.52 (N)	8.38E-06 $GP_O > DT$	0.22 (L)	0.45 $GP_O \approx DR$	0.46 (N)	0.002 $GP_O > Ensemble$	0.32 (M)
0.9	0.002 $GP_T > GP_O$	0.64 (M)	0.50 $GP_O \approx GP_N$	0.56 (S)	0.80 $GP_O \approx DT$	0.47 (N)	0.18 $GP_O \approx DR$	0.59 (S)	0.56 $GP_O \approx Ensemble$	0.54 (N)
0.95	0.0003 $GP_T > GP_O$	0.65 (M)	0.44 $GP_O \approx GP_N$	0.59 (S)	0.83 $GP_O \approx DT$	0.52 (N)	0.39 $GP_O \approx DR$	0.57 (S)	0.26 $GP_O \approx Ensemble$	0.59 (S)
1	6.99E-05 $GP_T > GP_O$	0.68 (M)	0.52 $GP_O \approx GP_N$	0.60 (S)	0.88 $GP_O \approx DT$	0.52 (N)	0.34 $GP_O \approx DR$	0.60 (S)	0.24 $GP_O \approx Ensemble$	0.61 (S)

Table B.4: Statistical test results comparing (a) the rate of pass verdicts predicted as fail achieved with DR against those of GP_T, GP_O, GP_N, DT and ensemble (b) the rate of fail verdicts predicted as pass achieved with GP_O against those of GP_T, GP_N, DT, DR and ensemble for each study subject. The p-values highlighted in blue represent cases where the method on the left significantly outperforms the compared alternative. The p-values highlighted in orange indicate cases where the compared alternative significantly outperforms the method on the left. The cells highlighted in yellow represent cases where DT is not applicable. The significance level is 0.05.

(a) Rate of pass verdicts predicted as fail

Study Subject	DR vs GP_T		DR vs GP_O		DR vs GP_N		DR vs DT		DR vs Ensemble	
	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$
Router	1.37E-49 $DR > GP_T$	0.13 (L)	4.95E-14 $DR > GP_O$	0.29 (L)	7.06E-17 $DR > GP_N$	0.27 (L)	1.18E-62 $DR > DT$	0.16 (L)	1.75E-58 $DR > Ensemble$	0.1 (L)
AP–DHB	2.34E-46 $DR > GP_T$	0.001 (L)	1.14E-24 $DR > GP_O$	0.01 (L)	5.19E-44 $DR > GP_N$	0 (L)	3.07E-67 $DR > DT$	0.03 (L)	8.41E-59 $DR > Ensemble$	0.01 (L)
AP–TWN (R1)	5.88E-38 $DR > GP_T$	0.19 (L)	1 $DR \approx GP_O$	0.5 (N)	6.48E-36 $DR > GP_N$	0.18 (L)	2.09E-59 $DR > DT$	0 (L)	9.9E-38 $DR > Ensemble$	0.19 (L)
AP–TWN (R2)	1 $DR \approx GP_T$	0.5 (N)	1.61E-21 $DR > GP_O$	0.31 (M)	2.35E-25 $DR > GP_N$	0.28 (L)	1 $DR \approx DT$	0.5 (N)	1 $DR \approx Ensemble$	0.5 (N)
AP–TWN (R3)	0.005 $DR > GP_T$	0.48 (N)	7.63E-63 $DR > GP_O$	0 (L)	1.2E-50 $DR > GP_N$	0 (L)	1 $DR \approx DT$	0.5 (N)	0.004 $DR > Ensemble$	0.47 (N)
AP–TWN (R4)	0.0009 $DR > GP_T$	0.30 (M)	0.0001 $DR > GP_O$	0.29 (L)	0.001 $DR > GP_N$	0.29 (L)	DT is not applicable		0.001 $DR > Ensemble$	0.30 (M)
AP–SNG	2.15E-13 $DR > GP_T$	0.27 (L)	6.34E-44 $DR > GP_O$	0.03 (L)	1.16E-45 $DR > GP_N$	0.03 (L)	2.42E-76 $DR > DT$	0.005 (L)	1.06E-71 $DR > Ensemble$	0.008 (L)
Dave2	1.23E-10 $DR > GP_T$	0.39 (S)	3.99E-42 $DR > GP_O$	0.09 (L)	3.11E-32 $DR > GP_N$	0.20 (L)	1 $DR \approx DT$	0.5 (N)	0.19 $DR \approx Ensemble$	0.49 (N)

(b) Rate of fail verdicts predicted as pass

Study Subject	GP_O vs GP_T		GP_O vs GP_N		GP_O vs DT		GP_O vs DR		GP_O vs Ensemble	
	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$
Router	1.88E-19 $GP_T > GP_O$	0.70 (M)	0.18 $GP_O \approx GP_N$	0.53 (N)	0.006 $GP_O > DT$	0.42 (S)	0.0007 $GP_O > DR$	0.40 (S)	9.37E-13 $Ensemble > GP_O$	0.67 (M)
AP–DHB	0.0007 $GP_O > GP_T$	0.33 (M)	0.04 $GP_O \approx GP_N$	0.40 (S)	0.02 $GP_O > DT$	0.42 (S)	3.2E-10 $DR > GP_O$	0.78 (L)	0.002 $GP_O > Ensemble$	0.36 (M)
AP–TWN (R1)	0.19 $GP_O \approx GP_T$	0.59 (S)	0.01 $GP_N > GP_O$	0.69 (M)	9.02E-12 $DT > GP_O$	1 (L)	5.06E-11 $DR > GP_O$	0.95 (L)	0.04 $Ensemble > GP_O$	0.64 (M)
AP–TWN (R2)	1.41E-06 $GP_T > GP_O$	0.66 (M)	0.24 $GP_O \approx GP_N$	0.54 (N)	1.82E-07 $DT > GP_O$	0.66 (M)	9.25E-36 $DR > GP_O$	0.81 (L)	3.4E-06 $Ensemble > GP_O$	0.66 (M)
AP–TWN (R3)	0.1 $GP_O \approx GP_T$	0.48 (N)	1 $GP_O \approx GP_N$	0.5 (N)	1 $GP_O \approx DT$	0.5 (N)	0.0008 $GP_O > DR$	0.44 (S)	0.0003 $GP_O > Ensemble$	0.43 (S)
AP–TWN (R4)	0.04 $GP_O > GP_T$	0.46 (N)	1 $GP_O \approx GP_N$	0.5 (N)	DT is not applicable		7.01E-14 $GP_O > DR$	0.14 (L)	0.005 $GP_O > Ensemble$	0.44 (S)
AP–SNG	0.002 $GP_O > GP_T$	0.45 (S)	1 $GP_O \approx GP_N$	0.5 (N)	1 $GP_O \approx DT$	0.5 (N)	2.61E-20 $GP_O > DR$	0.22 (L)	1 $GP_O \approx Ensemble$	0.5 (N)
Dave2	0.002 $GP_O > GP_T$	0.39 (S)	3.14E-05 $GP_O > GP_N$	0.34 (M)	5.33E-11 $GP_O > DT$	0.27 (L)	6.38E-07 $GP_O > DR$	0.31 (M)	4.78E-08 $GP_O > Ensemble$	0.29 (L)

Table B.5: Statistical tests comparing the relative accuracy results of DR against those of GP_T, GP_O, GP_N, DT and ensemble. The p-values highlighted in blue represent cases where DR significantly outperforms the compared alternative. The significance level is 0.05.

θ	**DR vs GP_T**		**DR vs GP_O**		**DR vs GP_N**		**DR vs DT**		**DR vs Ensemble**	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
0.5	3.31E-07 $DR > GP_T$	0.66 (M)	5.89E-08 $DR > GP_O$	0.68 (M)	2.38E-09 $DR > GP_N$	0.70 (M)	0.001 $DR > DT$	0.60 (S)	1.33E-08 $DR > Ensemble$	0.68 (M)
0.55	1.39E-05 $DR > GP_T$	0.64 (M)	1.83E-07 $DR > GP_O$	0.67 (M)	3.4E-10 $DR > GP_N$	0.71 (L)	8.32E-05 $DR > DT$	0.62 (S)	2.37E-07 $DR > Ensemble$	0.66 (M)
0.6	2.15E-05 $DR > GP_T$	0.63 (S)	3.4E-05 $DR > GP_O$	0.64 (M)	6.58E-10 $DR > GP_N$	0.70 (M)	8.32E-05 $DR > DT$	0.62 (S)	2.36E-07 $DR > Ensemble$	0.66 (M)
0.65	2.09E-06 $DR > GP_T$	0.65 (M)	0.001 $DR > GP_O$	0.62 (S)	1.13E-10 $DR > GP_N$	0.72 (L)	0.0009 $DR > DT$	0.61 (S)	2.98E-07 $DR > Ensemble$	0.66 (M)
0.7	9.62E-06 $DR > GP_T$	0.64 (M)	0.07 $DR \approx GP_O$	0.57 (S)	2.79E-06 $DR > GP_N$	0.67 (M)	0.0003 $DR > DT$	0.61 (S)	8.32E-08 $DR > Ensemble$	0.67 (M)
0.75	2.89E-06 $DR > GP_T$	0.66 (M)	0.04 $DR > GP_O$	0.59 (S)	0.005 $DR > GP_N$	0.61 (S)	5.57E-09 $DR > DT$	0.69 (M)	1.54E-13 $DR > Ensemble$	0.74 (L)
0.8	0.0002 $DR > GP_T$	0.62 (S)	5.82E-08 $DR > GP_O$	0.80 (L)	8.56E-05 $DR > GP_N$	0.66 (M)	1.38E-12 $DR > DT$	0.74 (L)	1.24E-14 $DR > Ensemble$	0.76 (L)
0.85	0.002 $DR > GP_T$	0.60 (S)	4.08E-09 $DR > GP_O$	0.85 (L)	2.92E-09 $DR > GP_N$	0.83 (L)	3.01E-15 $DR > DT$	0.76 (L)	1.57E-14 $DR > Ensemble$	0.76 (L)
0.9	0.01 $DR > GP_T$	0.59 (S)	2.88E-05 $DR > GP_O$	0.88 (L)	6.35E-06 $DR > GP_N$	0.86 (L)	1.19E-09 $DR > DT$	0.70 (M)	1.68E-08 $DR > Ensemble$	0.69 (M)
0.95	0.09 $DR \approx GP_T$	0.56 (N)	0.0007 $DR > GP_O$	0.88 (L)	0.0002 $DR > GP_N$	0.84 (L)	0.0004 $DR > DT$	0.62 (S)	0.006 $DR > Ensemble$	0.59 (S)
1	0.04 $DR > GP_T$	0.57 (S)	0.001 $DR > GP_O$	0.87 (L)	0.004 $DR > GP_N$	0.79 (L)	4.59E-05 $DR > DT$	0.66 (M)	0.01 $DR > Ensemble$	0.59 (S)

Table B.6: Statistical tests comparing the relative accuracy results of DR against those of GP_T, GP_O, GP_N, DT and ensemble for each study subject. The p-values highlighted in blue represent cases where ***DR*** significantly outperforms the compared alternative. The p-values highlighted in orange indicate cases where the compared alternative outperforms ***DR***. The cells highlighted in yellow represent cases where DT is not applicable. The significance level is 0.05.

Study Subject	**DR vs GP_T**		**DR vs GP_O**		**DR vs G_N**		**DR vs DT**		**DR vs Ensemble**	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
Router	0.0001 *$GP_T > DR$*	0.40 (S)	6.72E-18 *$DR > GP_O$*	0.74 (L)	4.3E-09 *$DR > GP_N$*	0.65 (M)	3.99E-14 *$DR > DT$*	0.65 (M)	2.6E-05 *Ensemble > DR*	0.39 (S)
AP–DHB	0.04 *$DR > GP_T$*	0.56 (S)	0.01 *$DR > GP_O$*	0.61 (S)	0.01 *$DR > GP_N$*	0.59 (S)	0.23 *$DR \approx DT$*	0.53 (N)	0.06 *$DR \approx$ Ensemble*	0.55 (N)
AP–TWN (R1)	3.43E-39 *$DR > GP_T$*	0.89 (L)	5.06E-11 *$DR > GP_O$*	0.95 (L)	1.85E-37 *$DR > GP_N$*	0.98 (L)	2.64E-34 *$DR > DT$*	1 (L)	4.35E-51 *DR > Ensemble*	0.97 (L)
AP–TWN (R2)	1.17E-49 *$DR > GP_T$*	0.85 (L)	7.52E-66 *$DR > GP_O$*	1 (L)	1.68E-66 *$DR > GP_N$*	1 (L)	2.36E-66 *$DR > DT$*	0.90 (L)	6.82E-62 *DR > Ensemble*	0.91 (L)
AP–TWN (R3)	0.08 *$DR \approx GP_T$*	0.47 (N)	2.74E-33 *$DR > GP_O$*	0.88 (L)	2.06E-21 *$DR > GP_N$*	0.88 (L)	2.4E-07 *$DR > DT$*	0.44 (S)	0.23 *$DR \approx$ Ensemble*	0.52 (N)
AP–TWN (R4)	2.02E-15 *$DR > GP_T$*	0.02 (L)	5.77E-21 *$DR > GP_O$*	0 (L)	1.31E-15 *$DR > GP_N$*	0 (L)	DT is not applicable		5.77E-10 *DR > Ensemble*	0.12 (L)
AP–SNG	8.78E-08 *$DR > GP_T$*	0.66 (M)	2.69E-40 *$DR > GP_O$*	0.94 (L)	6.89E-42 *$DR > GP_N$*	0.94 (L)	8.92E-62 *$DR > DT$*	0.94 (L)	2.97E-58 *DR > Ensemble*	0.94 (L)
Dave2	0.002 *$DR > GP_T$*	0.59 (S)	1.11E-10 *$DR > GP_O$*	0.74 (L)	9.29E-13 *$DR > GP_N$*	0.72 (L)	1.38E-31 *$DR > DT$*	0.82 (L)	3.73E-12 *DR > Ensemble*	0.70 (L)

Table B.7: Statistical test results comparing the accuracy of GP_O against GP_T, GP_N, DT, DR, and the ensemble when datasets TS_1 to TS_{10} are used. The p-values highlighted in blue indicate cases where GP_O significantly outperforms the alternative it is being compared to. The significance level is 0.05.

θ	GP_O **vs** GP_T		GP_O **vs** GP_N		GP_O **vs DT**		GP_O **vs DR**		GP_O **vs Ensemble**	
	P-value	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**P-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
0.5	1E-23 $GP_O > GP_T$	0.94 (L)	2.15E-05 $GP_O > GP_N$	0.60 (S)	3.29E-20 $GP_O > DT$	0.92 (L)	1.47E-25 $GP_O > DR$	0.98 (L)	2.74E-21 $GP_O > Ensemble$	0.89 (L)
0.55	3.73E-23 $GP_O > GP_T$	0.94 (L)	3.29E-06 $GP_O > GP_N$	0.61 (S)	2.22E-19 $GP_O > DT$	0.91 (L)	3.71E-25 $GP_O > DR$	0.97 (L)	4.74E-21 $GP_O > Ensemble$	0.88 (L)
0.6	1.23E-21 $GP_O > GP_T$	0.94 (L)	3.83E-06 $GP_O > GP_N$	0.66 (M)	6.28E-18 $GP_O > DT$	0.91 (L)	2.63E-23 $GP_O > DR$	0.97 (L)	1.29E-19 $GP_O > Ensemble$	0.89 (L)
0.65	7.65E-20 $GP_O > GP_T$	0.95 (L)	8.94E-06 $GP_O > GP_N$	0.70 (M)	5.28E-16 $GP_O > DT$	0.90 (L)	1.47E-20 $GP_O > DR$	0.96 (L)	5.07E-17 $GP_O > Ensemble$	0.90 (L)
0.7	5.14E-18 $GP_O > GP_T$	0.97 (L)	1.97E-06 $GP_O > GP_N$	0.78 (L)	2.79E-14 $GP_O > DT$	0.93 (L)	1.13E-17 $GP_O > DR$	0.96 (L)	8.92E-15 $GP_O > Ensemble$	0.92 (L)
0.75	8.1E-14 $GP_O > GP_T$	0.98 (L)	5.34E-08 $GP_O > GP_N$	0.86 (L)	3.37E-12 $GP_O > DT$	0.94 (L)	8.78E-14 $GP_O > DR$	0.98 (L)	7.87E-13 $GP_O > Ensemble$	0.95 (L)
0.8	1.1E-11 $GP_O > GP_T$	0.98 (L)	1.71E-07 $GP_O > GP_N$	0.86 (L)	1.61E-10 $GP_O > DT$	0.94 (L)	7.45E-12 $GP_O > DR$	0.98 (L)	2.81E-11 $GP_O > Ensemble$	0.96 (L)
0.85	5.21E-10 $GP_O > GP_T$	0.99 (L)	5.49E-06 $GP_O > GP_N$	0.84 (L)	2.06E-09 $GP_O > DT$	0.97 (L)	2.9E-10 $GP_O > DR$	0.99 (L)	5.21E-10 $GP_O > Ensemble$	0.99 (L)
0.9	2.1E-04 $GP_O > GP_T$	0.99 (L)	0.01 $GP_O > GP_N$	0.89 (L)	0.001 $GP_O > DT$	0.95 (L)	0.0001 $GP_O > DR$	0.99 (L)	1.39E-06 $GP_O > Ensemble$	0.97 (L)
0.95	0.00021 $GP_O > GP_T$	0.99 (L)	0.01 $GP_O > GP_N$	0.82 (L)	0.001 $GP_O > DT$	0.92 (L)	0.0001 $GP_O > DR$	0.99 (L)	0.0002 $GP_O > Ensemble$	0.95 (L)
1	0.009 $GP_O > GP_T$	0.92 (L)	0.08 $GP_O \approx GP_N$	0.83 (L)	0.02 $GP_O > DT$	0.91 (L)	0.003 $GP_O > DR$	0.99 (L)	0.005 $GP_O > Ensemble$	0.95 (L)

Appendix C

Prompt template for RQ4 in Chapter 6

Figure C.1 presents the outline of the prompt used to identify relevant sentences from the reference documentation for an assertion in RQ4 in Chapter 6.

Context: {System description}

Task: Given a natural-language assertion and a collection of documents, your task is to identify and return sentences that address the same condition, behaviour, or outcome as the assertion.

Inputs:

- Assertion: {Assertion text}
- Documentation: {List of documents}

Instructions:

1. Retrieve between 2 and 5 sentences from the documentation that best support the assertion. Return 2–3 sentences if only a few are highly related. Return 4–5 sentences if several are strongly related.
2. Exclude sentences that are loosely related or tangential.
3. Preserve the original sentence wording exactly as it appears in the document.
4. Output only the selected sentences, each on a new line; no explanations or metadata.

Example: {One-shot example}

Figure C.1: Our prompt template