Delta Debugging for Cyber-Physical Systems with Flaky Test Executions

PABLO VALLE, Mondragon University, Spain
SHAUKAT ALI, Simula Research Laboratory, Norway
AITOR ARRIETA, Mondragon University, Spain

Simulation-based testing is widely used to validate Cyber-Physical Systems (CPSs), yet modern CPS simulators frequently exhibit non-deterministic ("flaky") behavior, making failures difficult to reproduce and debug. Although delta debugging has proven effective for deterministic systems, its underlying assumptions do not hold in stochastic environments. This paper presents three delta debugging algorithms that combine statistical failure analysis, repeated executions, and environment-aware reduction to isolate minimal failure-inducing test inputs for stochastic CPSs. We evaluate the proposed techniques on two complementary case study systems: an industrial elevator dispatching system employing stochastic optimization and an autonomous mobile robot exhibiting simulator-induced non-determinism. The results show that the proposed approaches substantially reduce debugging time while preserving the original failure behavior. More importantly, we observe that minimizing failure-inducing test inputs frequently increases failure reproducibility compared with the original executions. By eliminating execution segments that introduce incidental stochastic effects, the reduced test inputs isolate the causal conditions of the failure and consistently reproduce it with higher probability. These findings suggest that delta debugging not only simplifies failure analysis but also mitigates execution flakiness, providing a practical foundation for debugging CPSs.

ACM Reference Format:
Pablo Valle, Shaukat Ali, and Aitor Arrieta. 2026. Delta Debugging for Cyber-Physical Systems with Flaky Test Executions. 1, 1 (July 2026), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cyber-Physical Systems (CPSs) integrate computational components with physical processes through sensing, communication, and control [2, 17, 27]. Accordingly, considerable research effort has been devoted to the verification and testing of CPSs, including simulation-based test generation [7, 8, 44, 45], regression test optimization [6, 9] mutation testing [19, 25, 41], and runtime monitoring [14, 54]. In comparison, considerably less attention has been paid to the automated debugging of failures once they have been detected. This imbalance is problematic because CPS failures commonly emerge from long interactions between software components, controllers, physical processes, and environmental conditions, leaving engineers with extensive execution traces and complex test inputs that are difficult to inspect manually.

Simulation-based testing is central to the development of CPSs because it enables engineers to exercise operational conditions that would be expensive, time-consuming, or unsafe to reproduce using physical prototypes. However, repeated executions of the same test input do not necessarily

Authors' Contact Information: Pablo Valle, pvalle@mondragon.edu, Mondragon University, Mondragon, Gipuzkoa, Spain; Shaukat Ali, shaukat@simula.no, Simula Research Laboratory, Oslo, Norway; Aitor Arrieta, aarrieta@mondragon.edu, Mondragon University, Mondragon, Gipuzkoa, Spain.

ACM XXXX-XXXX/2026/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

produce identical outcomes. Sources of stochasticity may originate from the System Under Test (SUT), the simulation infrastructure, or their interaction. For example, CPS controllers may employ randomized optimization algorithms, such as genetic algorithms, while simulators may introduce variability through physics computations, floating-point operations, concurrency, and event scheduling. Additional non-determinism may arise from perception pipelines, variable inference times, and communication middleware such as the Robot Operating System (ROS). Consequently, an identical test input may produce different trajectories, quantitative outcomes, or even different pass/fail verdicts across repeated executions.

Recent empirical evidence indicates that such flakiness is not exceptional. Amini et al. [3], for example, observed substantial variation across several autonomous-driving test configurations, with soft flakiness affecting between 4% and 68% of the generated tests and hard flakiness reaching up to 74% in some configurations. They further showed that rerunning test inputs can materially change both the fitness values obtained by randomized testing algorithms and the number of failures detected. These findings demonstrate that simulator flakiness can undermine not only testing outcomes but also any subsequent debugging activity that assumes deterministic failure reproduction. A reduced input may appear to preserve a failure in one execution and fail to reproduce it in the next, making it difficult to determine whether the reduction removed a necessary condition or whether the observed difference is merely caused by stochastic execution variability.

One established strategy for facilitating debugging is to reduce a failure-inducing test input to a smaller input that still triggers the original failure. Delta Debugging [65] systematically removes parts of an input and repeatedly executes the resulting candidates to isolate a minimal failure-inducing subset. In CPSs, this can substantially reduce the amount of information that engineers must inspect. For example, a full-day elevator traffic scenario containing thousands of passenger events may be reduced to the comparatively small set of events immediately preceding an abnormal dispatching decision [60]. Similarly, a long robotic trajectory may be reduced to the segment required to reproduce a lane-departure failure. However, conventional Delta Debugging assumes that the outcome of each candidate input can be determined from a single execution. This assumption does not hold for stochastic CPSs.

A direct adaptation would be to execute every candidate input multiple times and statistically determine whether it preserves the original failure. Although this strategy improves confidence in the reduction decision, it can make Delta Debugging prohibitively expensive. CPS test executions often involve computationally demanding simulators and long-running scenarios [6, 9], and Delta Debugging already requires the evaluation of multiple candidate inputs. Repeating every candidate many times therefore multiplies an already substantial computational cost. The central challenge is thus to preserve the statistical reliability required by stochastic executions while avoiding unnecessary test reruns.

To address this challenge, we first establish a stochastic Delta Debugging baseline that replaces the deterministic pass/fail decision with repeated executions, failure clustering, and statistical comparison. While this baseline enables reliable reduction under execution variability, its computational cost is prohibitive for long-running CPS simulations. We therefore propose two techniques that improve its practicality. Optimized Stochastic Delta Debugging (DD_{OS}) reduces execution time by speculatively evaluating candidate reductions, storing intermediate solutions, and performing repeated statistical validation only when necessary. Environment-Wise Optimized Stochastic Delta Debugging ($EWDD_{OS}$) further exploits stable states of the CPS environment to identify promising starting points for reduction, further accelerating the debugging process. Instead of requiring identical execution outcomes, both techniques determine whether a candidate reproduces the same class and statistical characteristics of the original failure. We evaluate the approaches using two complementary CPSs (an industrial CPS case study system and an open-source case study system).

Our results show that the proposed techniques can substantially reduce failure-inducing test inputs and that the optimized variants can decrease debugging cost compared with conservatively validating every reduction. More importantly, the results reveal that minimization increases the failure reproduction ratio of the resulting test inputs, thereby reducing flakiness. Thus, the proposed techniques produce test inputs that are not only smaller, but often more reliable debugging artifacts.

The main contributions of this paper are:

- We formulate Delta Debugging for CPSs tested under test flakiness, replacing deterministic failure preservation with a statistical assessment based on repeated executions, failure clustering, and distributional comparison.
- We formulate a stochastic baseline for Delta Debugging based on repeated executions and statistical failure comparison, and propose two practical extensions that reduce its computational cost through speculative validation and environment-aware reduction.
- We conduct an empirical evaluation on an industrial elevator-dispatching system and a DNN-controlled autonomous mobile robot, covering multiple scenarios and distinct sources of stochasticity.
- We show that the proposed techniques effectively reduce failure-inducing inputs while decreasing the execution cost of debugging. Most notably, we find that minimized inputs are able to reproduce the target failure more reliably than the original inputs, suggesting that test-input reduction can mitigate the effects of execution flakiness.
- We provide a replication package including both a virtual machine with the ROS environment for reproducing the experiments for the Leo Rover case study [58] and the scripts fro analyzing the results from both case studies [57].

The rest of the paper is structured as follows: We give a general background of testing and failure isolation in the context of CPSs in Section 2. Section 3 introduces our industrial and open-source case studies and which are the sources of randomness on each of them. In Section 4 we present our approach. In Sections 5 and 6 we present our empirical evaluation and we analyze the results. We draw conclusions from the results and highlight key lessons learned in Section 7. Finally, in Section 8 we position our work with existing works and we conclude the paper presenting our future work in Section 9.

2 BACKGROUND

This section introduces key aspects related to testing and failure isolation in the context of CPSs.

2.1 Simulation-based testing

Simulation-based testing provides a versatile environment to test CPSs across different development stages. In this context, testing is structured into different levels that progressively increase the fidelity of the simulation environment. At the highest level of abstraction, model-in-the-loop (MIL) testing [5] emphasizes on verifying the correctness of mathematical models and control algorithms. This is succeed by software-in-the-loop (SIL) testing [44], in which the real control software operates within a simulated setting to assess its integration with dynamic models. This is the level in which this study is developed. Finally, at the lowest level of abstraction, hardware-in-the-loop (HIL) testing [30, 38] combines physical hardware elements with simulation models to accurately replicate real-world interactions. This tiered approach helps identify and address possible problems early in the development process, balancing development costs, risks, and accuracy.

As simulation-based testing allows engineers to represent both the physical dynamics and the control software of the CPS [4, 37, 39, 54], it facilitates the exploration of a wide range of operational scenarios without the need for expensive physical prototypes. Tools like Simulink, Modelica, and

Gazebo are commonly employed to model the dynamic behavior of physical processes and their interactions with CPS controllers. This method not only reduces the development costs and risks but also allows for the simultaneous execution of large-scale experiments, which is crucial due to the wide input space of CPSs [21, 44, 46, 48].

However, as simulators become increasingly realistic, ensuring deterministic executions becomes progressively more challenging [3]. Small differences in floating-point computations, numerical integration, event scheduling, thread interleavings, and timing accumulate throughout the simulation, causing identical test inputs to produce different execution traces and, in some cases, different failure outcomes. This execution variability makes failures difficult to reproduce and considerably complicates debugging [3]. Consequently, debugging techniques for modern CPSs should explicitly account for stochastic execution behavior rather than assuming deterministic test outcomes.

2.2 Flaky Simulators

Deterministic simulation has historically served as the foundation for testing CPSs, providing a framework for reproducible evaluation under controlled conditions [10, 22, 39]. Simulators like Carla (developed using Unreal Engine), MetaDrive and Gazebo are specifically designed to provide virtual settings where environmental factors can be accurately controlled. This control is essential for replicating scenarios that would be dangerous or impractical to evaluate in the physical world. In theory, these simulators facilitate reliable testing by guaranteeing that multiple executions of a test input produce the same behavior of the SUT by seed controlled randomization and the deterministic nature of the physics engines. Such determinism is crucial not only for simple testing techniques but also for sophisticated testing methods such as regression testing [42], mutation testing [25], safety validation [26] and test case generation [8].

However, as pointed out by Osikowicz et al. [49] and Amini et al. [3] various commonly used CPS simulators face significant non-determinism. This is due to several factors such as artifacts of the physics engine, discrete-time integration techniques and floating-point estimations in numerical solvers. In addition, the physics computation and enhancements to the rendering of the images in such simulators also could lead to non-deterministic behavior. Concurrency also contributes to the occurrence of flaky simulation results.

In this work, we consider two complementary case study systems that exhibit stochastic behavior for different reasons. The first is an industrial elevator-dispatching system developed by Orona. Although the simulation environment itself is deterministic, the System Under Test (SUT) employs a Genetic Algorithm to optimize elevator assignments in large search spaces. Consequently, repeated executions of the same test input may produce different dispatching decisions and, therefore, different system behaviors. The second case study system is an autonomous mobile robot based on the Leo Rover platform[1]. In this case, stochasticity primarily originates from the execution platform rather than the controller itself. The robot is evaluated in the Gazebo simulator and relies on the Robot Operating System (ROS) for communication between its software components. Variability introduced by the physics engine, floating-point computations, event scheduling, image acquisition and processing, execution-time fluctuations of the DNN-based controller, and communication delays in ROS may cause identical test inputs to produce different execution traces and failure outcomes. Together, these two case study systems allow us to evaluate the proposed techniques under two distinct classes of stochasticity: algorithm-induced randomness and infrastructure-induced execution variability.

[1] https://www.leorover.tech/

2.3 Debugging of CPSs

Debugging Cyber-Physical Systems (CPSs) is inherently more challenging than debugging conventional software because faults often emerge from the interaction between software controllers, communication middleware, sensors, actuators, and the physical environment [55, 64]. Consequently, the root cause of a failure may not correspond to a single software defect, but instead to a particular sequence of events or environmental conditions that collectively drive the system into an unsafe state. This challenge is further exacerbated in stochastic CPSs, where identical test inputs may lead to different execution traces due to randomized algorithms, simulator artifacts, timing variations, or communication delays. As a result, failures can be difficult to reproduce consistently, substantially increasing the cost of debugging.

A major obstacle during debugging is the size of the execution that precedes a failure. CPS test inputs frequently consist of long event sequences or trajectories [60]. For example, in our industrial case study, a single simulation may represent an entire day of passenger traffic [60], comprising thousands of passenger arrivals and elevator movements before a failure is observed (Section 3.1). Identifying which subset of these events is actually responsible for the failure is therefore a difficult and time-consuming task.

Delta Debugging [65] addresses this problem by systematically reducing a failure-inducing test input while preserving the original failure. By removing execution segments that are not required to trigger the failure, the resulting minimized input substantially reduces the amount of information that engineers must inspect during fault localization. Our previous work [60] demonstrated the effectiveness of Delta Debugging for deterministic elevator dispatching systems. However, its underlying assumption that the same test input always reproduces the same failure no longer holds for stochastic CPSs, motivating the techniques proposed in this paper.

2.4 Delta Debugging

Delta Debugging, introduced by Zeller and Hildebrandt [65], is an automated input-reduction technique for isolating the elements of a test input that are necessary to reproduce a failure. Let t denote an input that triggers a failure. Delta Debugging repeatedly partitions t into smaller subsets and evaluates both these subsets and their complements. Whenever a smaller candidate still reproduces the failure, that candidate becomes the new input to be minimized. If neither the subsets nor their complements preserve the failure, the algorithm increases the partition granularity and continues the search. This process terminates when no individual element can be removed without losing the failure, yielding a *1-minimal* failure-inducing input. A 1-minimal input is not necessarily the globally smallest possible input, but none of its individual elements can be removed while preserving the observed failure.

By eliminating input elements that are not required to trigger the failure, Delta Debugging reduces the amount of information that developers must inspect during fault localization. It has consequently been applied to a broad range of failure-inducing artifacts, including tree-structured inputs such as XML documents [11, 47, 63], requests and interactions in microservice systems [66], formulas processed by Satisfiability Modulo Theories solvers [23], and test inputs for Cyber-Physical Systems [59, 60]. Although the structure of these inputs differs, the underlying principle remains the same: progressively remove parts of the original input while retaining the failure of interest.

In CPSs, the inputs subjected to reduction frequently consist of long sequences of operational events, commands, trajectories, or environmental conditions rather than conventional software artifacts. For example, a test of an elevator-dispatching system may encode an entire day of operation, including thousands of passenger arrivals, destination requests, and elevator movements [15, 16, 60, 61]. When such a test reveals a failure, only a small subset of those events may

be necessary to trigger it. Applying Delta Debugging can therefore produce a substantially shorter scenario that is easier to replay, inspect, and analyze.

Our previous work [60] adapted Delta Debugging to the CPS domain through an environment-aware reduction strategy for elevator-dispatching systems. The approach observes the state of the simulated environment and identifies stable operational states from which the reduction process can be started. By avoiding the repeated simulation of prefixes that do not influence the failure, the method can reduce both the size of the resulting test input and the computational cost of the reduction process.

However, classical Delta Debugging and our previous environment-aware adaptation assume deterministic test executions. Under this assumption, a candidate input either consistently reproduces the target failure or consistently does not. This binary decision is fundamental to the reduction process, as each observation determines which parts of the input are retained or discarded. In stochastic CPSs, however, repeated executions of the same candidate may produce different traces or failure outcomes. A candidate may reproduce the target failure in some executions but not in others, making a single pass/fail observation insufficient and potentially causing the algorithm to discard relevant input elements or retain irrelevant ones. Addressing this limitation requires replacing deterministic failure preservation with repeated execution and statistical comparison, while controlling the substantial computational cost introduced by such validation. This challenge motivates the stochastic Delta Debugging techniques presented in this paper.

3 CASE STUDY SYSTEMS

In this paper, we consider both an industrial case study system and an open-source case study system.

3.1 Industrial Case Study System – Elevator Dispatching Algorithm

3.1.1 Overview of the CPS. Figure 1 shows an overview of our industrial case study system provided by Orona [2], one of the worldwide leading companies in the elevation domain. This system involves a complex CPS where different computational units, communication protocols and mechanical and electrical components interact among them to transport passengers from a floor to another. Every time a passenger enters the system, the passenger makes a call through a button. This call is communicated to the dispatching algorithm through a Controller Area Network (CAN) bus. The dispatching algorithm determines which elevator is attending each call. This selection is based on different criteria, such as reducing passengers' waiting times or reducing energy consumption. When the dispatching algorithm selects which elevator should be assigned to a call, this is communicated through the CAN bus to the elevator controller. This controller makes the necessary moves to safely attend to the queued passengers.

3.1.2 The System Under Test. In our industrial case study, the System Under Test (SUT) is the traffic dispatching algorithm, which is the one that selects the elevator to attend each call. While Orona has a large suite of dispatching algorithms, we used one whose core algorithm is a Genetic Algorithm Beamurgia et al. [20] (as it is a stochastic algorithm, and therefore a good subject for our study). The dispatching algorithm takes information about the environment, such as the number of passengers each elevator has or the position of each elevator. Based on it, the algorithm returns a solution to assign an elevator to each call. However, the algorithm may operate with incomplete information, for example, the destination of the passenger, the weight of the passengers, or the number of passengers behind each call. Additionally, there may be cases where the system faces

[2] https://www.orona-group.com/es-es/

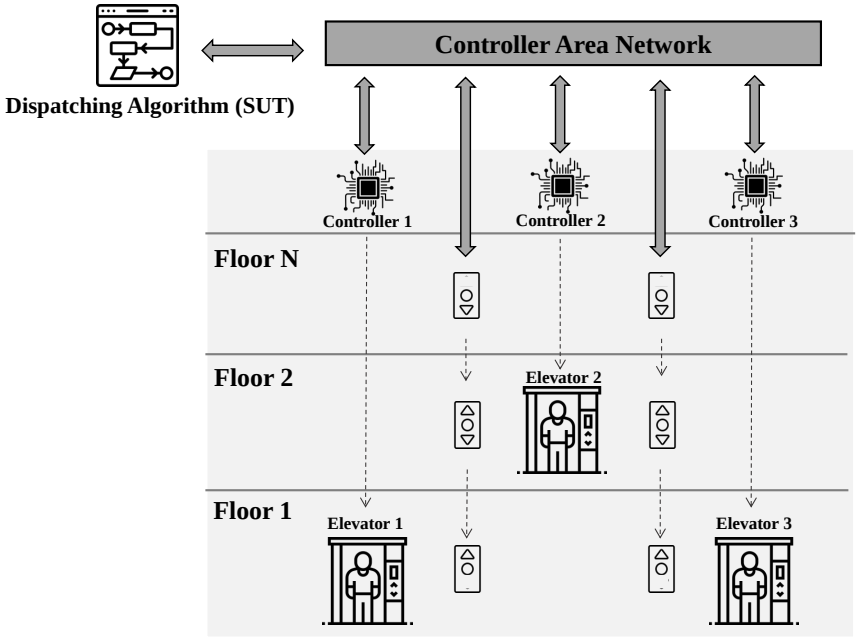


Fig. 1. Overview of our industrial case study

unforeseen situations for which the algorithm is not prepared or configured for [34, 35, 61]. When that occurs, the situation should be isolated as much as possible to propose a patch or adjustment.

3.1.3 Test Execution Platform. To test the traffic dispatching algorithms, different simulation test levels are conducted in Orona [13]. The first level refers to the Software-in-the-Loop (SiL) test level, where we integrate our technique. In such level, the traffic dispatching algorithm is integrated with Elevate[3], a commercial simulation tool. Elevate takes as input two files: (1) the installation of the elevators (including data like the number of elevators, speed of each of them, number of floors, etc.) and (2) the test input, which includes a set of passengers traveling through the installation. With these two input files, Elevate simulates the physical components of the elevator system (e.g., elevator speed, engines) and provides a file with several pieces of information (e.g., the time each passenger had to wait, which elevator attended each passenger, energy consumption). Then, an oracle parses this file to raise a test verdict (i.e., pass or fail). Orona also conducts tests in later test levels, including at the Hardware-in-the-Loop (HiL) test level as well as in operation. However, since our approach does not include this level, we do not explain it in detail. The reader can refer to a previous study to better understand the entire testing process conducted by Orona to test their dispatching algorithms [13].

3.1.4 Sources of randomness. The key source of randomness in this industrial case study system is its algorithm. The selected traffic dispatching algorithm is a genetic algorithm, which is employed in buildings with a large number of floors and where a large number of passengers is expected. Genetic algorithms are stochastic in nature, from its initial population generator, to latter genetic operators (e.g., mutation, crossover, selection). In this case, the simulator (i.e., Elevate), is deterministic, as all

[3]https://elevate.helpdocsonline.com/home

the communication between the dispatching algorithm and the rest of controllers (e.g., individual elevator controllers) are simulated.

3.2 Open-source Case Study System – Autonomous robot based on the Leo Rover platform

3.2.1 Overview of the CPS. Figure 2 provides an overview of our open-source case study system, the LeoRover [4] mobile robot. The Leo Rover is a mobile autonomous robot where different computational units, communication protocols, and mechanical and electrical components interact to enable effective navigation and task execution. The rover captures images that are processed by a Deep Neural Network (DNN) model. This model provides two outputs: (1) the linear velocity reference and (2) the angular velocity reference. These two outputs are communicated to a lower-level controller for translating them into engine speeds through a Robot Operating System (ROS)-based communication system. ROS is responsible for managing real-time interactions between the components and ensures that the commands are executed effectively and timely, allowing the rover to navigate its environment based on specific criteria (e.g., obstacle avoidance, path optimization, and task completion).

Fig. 2. LeoRover in our laboratory

3.2.2 The System Under Test. . For our open-source case study system, the SUT is the entire Leo Rover itself, an autonomous robotic platform composed of multiple interconnected subsystems, including a Deep Neural Network (DNN)–based navigation controller. Although the Leo Rover is designed to operate reliably under normal conditions and can adapt to previously unseen scenarios through its DNN-based controller, it remains susceptible to failures. These failures can stem from poorly trained DNN models, incorrect sensor calibration, or unexpected environmental variations. In such cases, isolating the failure is crucial to determine which component of the Leo Rover is responsible for the observed error and effectively identify the root cause.

3.2.3 Test Execution Platform. Leo Rover's control algorithm is connected to a simulation environment, specifically the Gazebo simulator [40]. Gazebo simulates the physical aspects of the rover's operation, including dynamics, sensor data, and interaction with the environment. Gazebo takes the environmental setup as the primary input, which includes information about the environment surroundings of the rover (e.g., terrain type, obstacles, lines to be followed by the rover, and the starting position and orientation of the rover). These inputs allow the simulator to replicate real-world conditions. The output from Gazebo includes data such as the rover's position, the

[4] https://github.com/LeoRover

path taken, and any collisions or errors found during navigation. This output is then analyzed using an oracle that verifies whether the rover is within the lines of the circuit. Based on this data, the algorithms proposed in this work are able to identify a failure and minimize the test inputs needed to reproduce that failure reliably, thus helping localize the source of the observed failure and identify the components resposible for these failures.

3.2.4 Sources of randomness. The behavior of the Leo Rover across simulators can vary due to three different sources of randomness: (1) the simulator, (2) the processing of images, and (3) the communication framework. Firstly, the Gazebo Simulator, which is crucial for our Software-in-the-Loop (SiL) testing, exhibits inherent non-determinism. Differences in its physics engine, event timing, and occasional unreliable performance in mimicking real-world dynamics create minor inconsistencies in sensor readings, collision detections, and general environmental interactions.

In addition, the image-based navigation system adds randomness. The time consumption to capture and process an image may vary across different runs because of different computational demands and delays in the navigation process. For instance, the image processing time of the DNN can vary due to differences in the system load and resource distribution. These timing variations can result in taking different images across runs, resulting in different outputs of the navigation system and different trajectories across runs. Furthermore, the ROS-based communication employed to manage data transfer between the Leo Rover's subsystems (i.e., sensors and controllers) may experience random delays. These delays, combined with the aforementioned randomness sources, make the Leo Rover behave stochastically.

4 APPROACH

In this section, we present our approach and its technical contributions.

4.1 Formalization

In the context of debugging of CPSs, let $\mathcal{T}$ represent all possible test inputs where each test input $TI \in \mathcal{T}$ consists of a collection or series of elements (e.g., passengers entering the building, guidance points), and let $f : \mathcal{TI} \rightarrow \{pass, fail\}$ be a test function that evaluates whether a test input causes a failure in the system under test (i.e., $f(TI) = fail$ indicates failure and $f(TI) = pass$ indicates correct behavior). Given an initial failure-inducing input TI_0 so that $f(TI_0) = fail$, the objective of Delta Debugging is to identify a subset $TI' \subseteq TI_0$ that is minimal with respect to its size while still inducing the same failure as TI_0. This means that there is no other subset $TI^* \subseteq TI_0$ which $|TI^*| < |TI'|$ and for which $f(TI^*) = fail$.

4.2 Running example

As a running example for Delta Debugging (Table 1), let us consider a Test input (TI). For the Leo Rover case study, the test input is composed of the starting point of the simulation and the circuit, which generates an output of 20 waypoints. Each waypoint (e.g., w_1, w_2, ..., w_{20}) includes key attributes like (1) timestamp, (2) coordinates (x,y) and (3) heading and (4) failure, which indicates whether the Leo Rover is outside the circuit ($failure \rightarrow 1$) or it is inside ($failure \rightarrow 0$). When the execution finishes, the oracle takes the output and evaluates whether a failure occurred, i.e., $f(TI)$ as *fail*. The objective of the Delta Debugging algorithm is to reduce this input (i.e., reduce the distance between the starting point of the simulation and the failure point) to a minimal subsequence of waypoints TI' so that the same failure still occurs, i.e., $f(TI') = fail$.

To do so, the algorithm starts by removing all the waypoints after the failure occurs since these waypoints don't affect the behavior of the SUT. Let's imagine that the Leo Rover went off the circuit at 8:10; consequently, w_{18} did not affect the failure, since the Leo Rover went out at w_{17},

when the failure was first detected. Therefore, the algorithm discards waypoints w_{18}, and w_{20} from the possible set of starting points. Then, the algorithm splits by half the sequence of remaining waypoints and executes the SUT and selects the last half of them as possible Starting points, i.e., $TI' = \{w_9, w_{10}, ..., w_{17}\}$, so that w_9 is the new starting point. That is, in the simulation, the rover starts at the position (X and Y) and heading (roll, pitch, yaw) of w_9. Suppose that the failure still persists with this TI'; this subset is then further divided by two, so that $TI' = \{w_{13}, w_{14}, w_{15}, w_{16}, w_{17}\}$ and the starting point is w_{13}. This process continues iteratively until the reduced test input is not able to reproduce the original failure, i.e., $f(TI') = Pass$. For instance, if $TI' = \{w_{15}, w_{16}, w_{17}\}$ is not able to reproduce the failure, a part of the removed waypoints (i.e., $\{w_{13}, w_{14}\}$) must be added. This process continues until the test input could not be reduced more while still reproducing the original failure. Thus, the minimal failure-inducing input is identified as a contiguous range of waypoints (e.g., $TI' = \{w_{14}, w_{15}, w_{16}, w_{17}\}$) and the starting point is w_{14}. This sequence could represent a sharp turn, a sudden change in speed, or conflicting heading directions that the CPS cannot handle correctly.

Table 1. Output waypoints for a Leo Rover navigating a circuit.

ID	Time	X (m)	Y (m)	Roll (rad)	Pitch (rad)	Yaw (rad)	Failure
w_1	08:00:00	0	0	0.00	0.00	1.57	0
w_2	08:01:00	5	0	0.00	0.00	1.57	0
w_3	08:01:30	10	0	0.00	0.00	1.57	0
w_4	08:01:45	15	0	0.00	0.00	1.57	0
w_5	08:02:20	20	0	0.00	0.00	1.57	0
w_6	08:03:25	20	5	0.00	0.00	0.00	0
w_7	08:03:50	20	10	0.00	0.00	0.00	0
w_8	08:04:35	15	10	0.00	0.00	4.71	0
w_9	08:05:40	10	10	0.00	0.00	4.71	0
w_{10}	08:06:15	5	10	0.00	0.00	4.71	0
w_{11}	08:06:40	0	10	0.00	0.00	4.71	0
w_{12}	08:07:55	0	5	0.00	0.00	3.14	0
w_{13}	08:08:30	0	0	0.00	0.00	3.14	0
w_{14}	08:08:45	5	0	0.00	0.00	1.57	0
w_{15}	08:09:10	10	0	0.00	0.00	1.57	0
w_{16}	08:09:25	15	0	0.00	0.00	1.57	0
w_{17}	08:10:20	20	0	0.00	0.00	1.57	1
w_{18}	08:11:00	20	-5	0.00	0.00	0.00	1
w_{19}	08:11:30	20	-10	0.00	0.00	0.00	1
w_{20}	08:12:00	15	-10	0.00	0.00	4.71	1

4.3 Delta Debugging for Stochastic Processes in CPSs

This section introduces the proposed delta debugging algorithm to deal with the stochastic nature of CPSs tested on flaky simulators. To this end, we implemented three versions of the algorithm: (1) Stochastic Delta Debugging, (2) Optimized Stochastic Delta Debugging, and (3) Environment-Wise Optimized Stochastic Delta Debugging. The former refers to an adaptation of the delta debugging algorithm for stochastic CPSs. In the second, we propose an enhanced version that addresses the challenge that CPS testing needs to deal with, i.e., long test execution time when needing to

minimize the failure inducing test input. In the latter, we add awareness of the environment to the delta debugging algorithm.

4.3.1 Stochastic Delta Debugging (DD_S). Algorithm 1 describes the Delta Debugging algorithm for CPSs tested on flaky simulators. As inputs, it receives the System Under Test (SUT), the initial failure inducing Test Input (TI) and its Failing Time (F_t), which indicates the simulation time at which the oracle detected the failure. As an output it provides, TI', which corresponds to the minimal failure-inducing Test Input for TI. This is obtained by following the next procedures.

First, the algorithm categorizes the failures into clusters (Line 1, Algorithm 1) to consider the chance of triggering different failures across the executions of the original test input. As Figure 3 depicts, the same test input may produce different behavior of the SUT and also trigger different failures. Similar to other research studies [12, 28, 29], we used clusters to identify different instances of failures. By means of ClusterFailures function, the algorithm selects the cluster with highest number of failures, assuming that the most frequently occurring failure is the most critical one. Then, the algorithm splits the test input to stop once the selected failure is triggered (Line 2, Algorithm 1). This is performed because if the SUT fails at certain point, there is no need to carry out executing the remaining test. For instance, if the rover goes off the road after 50 seconds, the algorithm does not need to continue the simulation once this event happens. Afterward, the algorithm calculates the reduction range to be applied for minimization and maximization purposes (Line 3), followed by the first reduction process (Line 4). This process is carried out by invoking Algorithm 2, which takes as input (i) TI' and (ii) *reduction*. Therefore, the function splits the test input TI' by half, returning the exact position of the rover (i.e., point $TI'_{n/2}$).

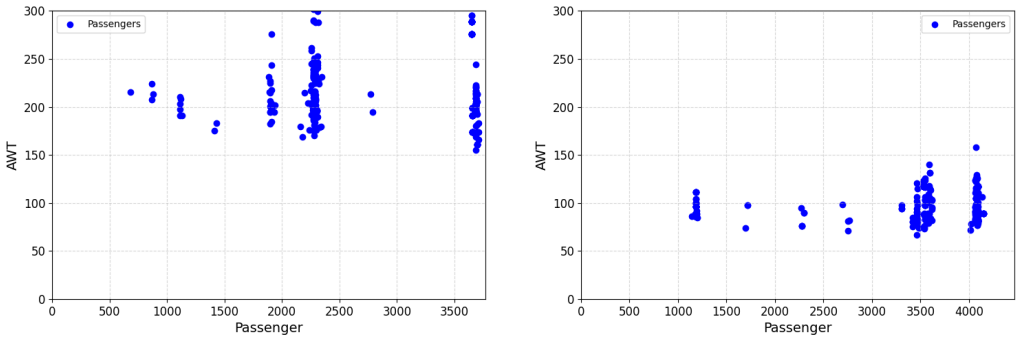


Fig. 3. Failure distribution across multiple runs in Scenario 1 and Scenario 6 for Orona's case study.

After the initial minimization, the algorithm enters a while loop (Lines 5-14) that aims to further minimize the failing test input. Within this loop, the test input TI_{NEW} is executed (Line 6) using the function ExecuteMultipleTimes, which runs TI_{NEW} a configurable number of times (i.e., 30 by default). This function evaluates whether there is a statistically significant difference between the failures induced by TI_{NEW} and the original failure.

The evaluation of executions first analyzes how many runs triggered the original failure by assessing how many failures belong to the selected cluster. Afterwards, the algorithm employs two statistical tests: Fisher's exact test [56] and the Mann-Whitney U test [43]. The decision to use both, Fisher's exact test and the Mann-Whitney U test, is based on the need for a robust assessment of the test input's reliability and the nature of the failures it reproduces. Fisher's exact test ensures consistency in the reproduction of failures, while the Mann-Whitney test verifies that the original failure is actually reproduced with statistical significance.

Algorithm 1: Stochastic Delta Debugging Algorithm

```
   Input: SUT ;                                          // System Under Test
          F_t ;                                                 // Failing time
          TI = {p_1, p_2, ..., p_n} ;          // Initial failure inducing test input
   Output: TI' = {p'_1, p'_2, ..., p'_n} ;     // Minimized failure inducing test input
1  cluster ← ClusterFailures(TI, F_t) ;
2  TI' ← Split(TI, F_t);
3  reduction ← TI'.np/2 ;
4  TI_NEW ← SplitMin(TI', reduction);
5  while reduction ≥ 1 do
6  |  Verdict ← ExecuteMultTimes(TI_NEW, SUT, cluster);
7  |  reduction ← reduction/2;
8  |  if Verdict == Failure then
9  |  |  TI' ← TI_NEW;
10 |  |  TI_NEW ← SplitMin(TI_NEW, reduction);
11 |  else
12 |  |  TI_NEW ← SplitMax(TI_NEW,TI', reduction);
13 |  end
14 end
```

Fisher's exact test is employed to determine whether TI_{NEW} consistently reproduces a failure. The algorithm generates a contingency matrix based on the clustering results of multiple runs. The goal is to assess whether there is no statistically significant difference in the failure rates between TI_{NEW} and TI. An odds ratio value lower than 1.68 [24] indicates statistical insignificance or there is statistical significance in favor of TI_{NEW} in a 95% of confidence level, suggesting that both the original test input and TI_{NEW} or TI_{NEW} is more likely to reproduce the failure, therefore TI_{NEW} is able to reproduce the failure consistently. This step is crucial since a test input that occasionally reproduces a failure is not considered reliable for further minimization. Once it is established that TI_{NEW} can consistently reproduce failures at a similar rate to TI, the Mann-Whitney U test is applied. This test evaluates whether the characteristics of the failures reproduced by TI_{NEW} and TI are statistically similar. A p-value greater than 0.05 indicates no significant difference between both test inputs, implying that TI_{NEW} effectively reproduces the original failure scenario.

Algorithm 2: SplitMinEvent: Split Minimizing

```
   Input: TI_NEW={p_1, p_2, ..., p_n} ;   // Minimized test input previously selected
          splitSize ;                                        // # of points to remove
   Output: TI_MIN ;                                           // Minimized test input
1  for i ← splitSize to TI_NEW.np do
2  |  TI_MIN ← TI_MIN ∪ p_i;
3  end
```

If the returned verdict is *"failure"*, the minimization procedure can continue (Lines 8-10), as it means that TI_{NEW} reproduces the original failure consistently. The test input in TI_{NEW} is assigned to TI' (Line 9), and the minimization routine is invoked by means of the *SplitMin* function (line 10).

Algorithm 3: SplitMax: Split Maximizing

```
   Input: TI_NEW ;                                  // Current test input
          TI' = {p_1, p_2, ..., p_n} ;              // Minimized test input
          splitSize ;                               // # of points to add
   Output: TI_MAX ;                                 // Maximized test input
1  toSplit ← TI.np-(TI_NEW.np + splitSize);
2  for i ← toSplit to TI.np do
3  |   TI_MAX ← TI_MAX ∪ p_i;
4  end
```

Conversely, if the returned verdict is *"pass"*, it means that TI_{NEW} does not reproduce the original failure, therefore, the test input requires to be enlarged (Lines 11-13), by invoking Algorithm 3 (Line 12), which takes as input (i) TI_{NEW}, (ii) TI' and (iii) $redeuction$. This Algorithm adds the reduction number of points to TI_{NEW} from TI', thus enlarging current TI_{NEW}. This procedure returns the maximized test input in TI_{NEW}, which is tested in Line 5, Algorithm 1. This process is repeated until TI' and TI_{NEW} are the same, i.e., the reduction is ≤ 1. When this condition is met, the algorithm returns the minimal failure-inducing test input.

4.3.2 Optimized Stochastic Delta Debugging (DD_{OS}). The optimized version of the Delta Debugging algorithm for CPSs tested on flaky simulators (i.e., Algorithm 4) differs from the normal version. This recursive algorithm iteratively refines an archive of candidate solutions until it identifies the optimal one. The algorithm can be divided into two components: (1) recursion handling (Algorithm 4, Lines 2–12) and (2) the minimization procedure (Algorithm 4, Lines 13-25).

First, the algorithm starts by determining whether it is running for the first time (i.e., the number of solutions in the archive is 0) or it is operating recursively on an archive that already contains multiple test inputs. In the initial run, the archive is empty, therefore it initializes the archive by adding the original test input to it as well as it splits the test input by the failing time (Line 11). This split is critical because it isolates the test input that is responsible for the failure, thereby setting the first stage for further reduction. Once the archive is initialized, the algorithm splits the test input by half (Line 13) and enters in the reduction procedure (Lines 14-24).

Within the reduction procedure, the algorithm first executes once the current test input by invoking the function ExecuteTest (Line 15). This function executes once the test input and assesses whether the failure is statistically similar to the ones obtained from TI'. It does this by first checking if the observed failure lies within the selected cluster of failures. Then, it computes the Z-score [62], which measures how many standard deviations the failure raised by TI_{NEW} is from the mean of the set of failures raised by TI'. This test determines how common the result is within the distribution. By using a Z-score range of -1.96 to 1.96, we established a confidence interval of 95%, which is a common statistical threshold, the same as the one used in the analysis of Fisher's exact test and Mann-Whitney U test. Therefore, if the Z-score falls within the range, it indicates that the failure raised by TI_{NEW} is similar to the one raised by TI' (i.e., the verdict is set to *"Failure"*).

On the contrary, if the verdict is *"Pass"*, the algorithm initiates a more rigorous verification process via the CheckMultTimes function (Line 17). This function first executes multiple times (i.e., by default 30 times) TI_{NEW} and assesses statistically whether the failure reproduced is similar to the original one by means of clustering, Fisher's exact test and Mann-Whitney U test as in Algorithm 1. When these additional tests continue to produce a *"Pass"* verdict, (i.e., TI_{NEW} is not

Algorithm 4: Optimized Delta Debugging Algorithm

Input: SUT ; // System Under Test
F_t ; // Failing time (Optional)
$TI = \{p_1, p_2, ..., p_n\}$; // Initial failure inducing test input
$archive$; // Archive of solutions (Optional)
$cluster$; // Cluster of failures
Output: $archive = \{TI_1, TI_2, ..., TI_n\}$; // Minimized failure inducing test inputs

1 $TI_{\text{NEW}} \leftarrow TI$;
2 **if** $archive.n > 0$ **then**
3 $TI_{\text{NEW}}, archive, reduction \leftarrow$ CheckSolutions($TI_{\text{NEW}}, archive, SUT, cluster$);
4 **if** $reduction == 0$ **then**
5 $archive \leftarrow$ saveToArchive($TI_{\text{NEW}}, 1$);
6 **return** $archive$;
7 **else**
8 $archive \leftarrow$ saveToArchive($TI_{\text{NEW}}, 1$);
9 **end**
10 **else**
11 $TI', archive, reduction \leftarrow$ InitArchive(TI, F_t);
12 **end**
13 $TI_{\text{NEW}} \leftarrow$ SplitMin($TI', reduction$);
14 **while** $reduction \geq 1$ **do**
15 $Verdict \leftarrow$ Execute($TI_{\text{NEW}}, SUT, cluster$);
16 **if** $Verdict \neq Failure$ **then**
17 $TI_{\text{NEW}}, archive, reduction \leftarrow$ CheckMultTimes($TI_{\text{NEW}}, archive, SUT, cluster$);
18 $archive \leftarrow$ SaveToArchive($TI_{\text{NEW}}, 1$);
19 **else**
20 $archive \leftarrow$ SaveToArchive($TI_{\text{NEW}}, 0$);
21 **end**
22 $reduction \leftarrow reduction/2$;
23 $TI_{\text{NEW}} \leftarrow$ SplitMin($TI_{\text{NEW}}, reduction$);
24 **end**
25 $archive \leftarrow$ OptimizedDeltaDebugging(TI_{NEW},$archive, SUT, cluster$) ;

able to reproduce the original failure consistently), the algorithm enters in a rollback phase (Lines 2-8, Algorithm 5). Inside this loop, the minimization is undone, specifically, the algorithm reverts the test input to a previous test input stored in the archive (Line 3, Algorithm 5). In addition, the algorithm updates the reduction, since TI_{NEW} is not able to reproduce the failure, it does not make sense to continue reducing, therefore, the failure-inducing minimal test input could be between the current TI_{NEW} and the discarded test input. So, in Line 3, the algorithm gets the difference (i.e., *reduction*) between the current TI_{NEW} and the discarded test input. If TI_{NEW} has already been statistically assessed, (i.e., $verified$ is *1*) the algorithm exits this loop (Lines 2-8). If not, TI_{NEW} is executed multiple times and statistically assessed (Line 7). This process is repeated until one of the test input in the archive statistically reproduces the failure raised by the original test input.

Algorithm 5: CheckMultTimes: Evaluate a test input multiple times

```
   Input: TI_NEW ;                                  // Test input Under Evaluation
   archive ;                                        // Archive of solutions
   SUT ;                                            // System Under Test
   cluster ;                                        // Cluster of failures
   Output: TI_NEW ;                                 // Minimized failure inducing test inputs
   archive ;                                        // Archive of solutions
   reduction ;                                      // Test input reduction range
1  statisticalVerdict ← ExecuteMultTimes(TI_NEW, SUT);
2  while statisticalVerdict == 0 do
3  |  TI_NEW, verified, archive, reduction ← UndoMinimization(TI_NEW, archive);
4  |  if verified==1 then
5  |  |  break;
6  |  else
7  |  |  statisticalVerdict ← ExecuteMultTimes(TI_NEW, SUT);
8  |  end
9  end
```

Once a candidate that reliably induces a failure is identified, the archive is updated accordingly (Lines 18-21). The algorithm saves in the archive the verified test input, in which a 1 indicates that the test input has been executed multiple times and a 0 indicates that the test input has only been executed once. This candidate is further reduced (Lines 22-23), and the process of execution, evaluation, and potential rollback is repeated until the reduction does not produce a significant change (i.e., when the reduction value becomes less than or equal to one). At this point, the test input is considered the minimal failure-inducing input, having reduced to the smallest possible size without losing its failure-inducing capacity.

After each successful reduction (i.e., the algorithm reaches line 25), the algorithm calls itself, using the current archive as input. This recursive invocation is essential to ensure that the latest candidate is re-evaluated in case it was only executed once. At this point, the algorithm instead of going directly to Line 11, it enters in the Lines 2-10 procedures. In this procedure, the algorithm proceeds by invoking the CheckSolutions function. The objective of this function is to verify that the candidate test input in the archive consistently reproduces the failure. To achieve this, the function first checks whether the current solution has already been assessed (Line 1, Algorithm 6). If so, the function returns the current solution as the minimized test input with a reduction value of zero. Conversely, the algorithm enters in a while loop (Lines 2-8). It can happen that the solution in the archive has not already been statistically assessed (i.e., executed several times), therefore it is executed multiple times (i.e., 30 times default) and compared to the original results (Line 7). This process continues until finding a solution in the archive that statistically reproduces the original failure (i.e, $verified$ is 1 or $statisticalVerdict$ is one). When this procedure finishes it can happen that between current solution and the previous one there is still a range of minimization (i.e.,$reduction > 0$) or there is no range of minimization (i..e, $reduction == 0$). In this case, the solution is saved to the archive (Line 5, Algorithm 4) and the algorithm returns the current archive. In the former case, the solution is saved to the archive and the minimization procedure starts again.

4.3.3 Environment-Wise Optimized Stochastic Delta Debugging ($EWDD_{OS}$). We now introduce the proposed Environment-Wise Optimized Delta Debugging algorithm ($EWDD_{OS}$)), presented

Algorithm 6: CheckSolutions: Check the solutions in the archive

```
   Input: TI_NEW ;                                  // Test input Under Evaluation
   archive ;                                        // Archive of solutions
   SUT ;                                            // System Under Test
   cluster ;                                        // Cluster of failures
   Output: TI_NEW ;                                 // Minimized failure inducing test inputs
   archive ;                                        // Archive of solutions
   reduction ;                                      // Test input reduction range
1  TI_NEW, verified, archive, reduction ← UndoMinimization(TI_NEW, archive);
2  while verified == 0 do
3  |  statisticalVerdict ← ExecuteMultTimes(TI_NEW, SUT);
4  |  if statisticalVerdict == 1 then
5  |  |  break;
6  |  else
7  |  |  TI_NEW, verified, archive, reduction ← UndoMinimization(TI_NEW, archive);
8  |  end
9  end
```

in Algorithm 7. This algorithm extends Algorithm 4 by incorporating a preprocessing phase (Algorithm 7 Lines 6-15) that handles static situations and generates an initial approximation of the minimized test input. Before the execution of Algorithm 7, the results from the original test executions are analyzed to identify clusters of failures and the corresponding static situations for each execution of the original test input. Then, all the static situations from all the executions inside the failing cluster (i.e., the cluster of the selected failure) are taken. Due to the inherent non-determinism of the CPS and Simulator, these static situations can vary across different runs, particularly in timing. To address this variability, the algorithm selects a common time slot that is shared among all identified static situations at a given points, as Figure 4 depicts.

Once this result analysis phase is completed, the execution of Algorithm 7 begins. If it is the first time that this algorithm is executed the archive is then initialized with the original test input (Line 5). If static situations are present the algorithm enters in a loop (Lines 7-16) that iteratively refines the test input to ensure that it still reproduces the original failure. In each iteration, the algorithm selects the last static situation before the failure (Line 8). Then in Lines 9 and 10, the algorithm reduces the test input and updates the environment according to the static situation (i.e., the elevator positions is adjusted). The reduced test input is then executed once. If that reduced test input is able to reproduce the original failure, then it is assessed multiple times (i.e., 30 times by default) so the results are statistically compared to the original results (Line 13). If the reduced test input consistently reproduces the original failure, the algorithm exits the loop and updates the archive with this reduced test input (Line 17). Otherwise, the algorithm iterates with the preceding static situation. After this minimization phase finishes, the algorithm proceeds with the next steps outlined in Algorithm 4 (Lines 13-25).

5 EMPIRICAL EVALUATION

This section empirically evaluates the proposed approach using an industrial and an open-source case study system. We aimed at answering the following research questions (RQs):

Algorithm 7: Environment-wise Optimized Delta Debugging Algorithm

Input: SUT ; // System Under Test
F_t ; // Failing time (Optional)
$TI = \{p_1, p_2, ..., p_n\}$; // Initial failure inducing test input
$archive$; // Archive of solutions (Optional)
$cluster$; // Cluster of failures
$staticSituations$; // Array of Static Situations
Output: $archive = \{TI_1, TI_2, ..., TI_n\}$; // Minimized failure inducing test inputs

1 $TI_{\text{NEW}} \leftarrow TI$;
2 **if** $archive.n > 0$ **then**
3 Lines 3-9 Algorithm 4
4 **else**
5 TI', $archive$, $reduction \leftarrow$ InitArchive(TI, F_t);
6 **if** $staticSituations.n > 0$ **then**
7 **while** $verdict == 0$ **do**
8 $currentStatic \leftarrow$ GetStaticSituation($staticSituations, F_t$);
9 $TI' \leftarrow$ ReduceTestInput($currentStatic$);
10 UpdateEnv($currentStatic$);
11 $Verdict \leftarrow$ Execute($TI_{\text{NEW}}, SUT, cluster$);
12 **if** $verdict == 1$ **then**
13 $verdict \leftarrow$ ExecuteMultTimes($TI', SUT, cluster$);
14 **end**
15 $F_t \leftarrow$ GetStaticTime($currentStatic$);
16 **end**
17 $archive \leftarrow$ SaveToArchive(TI', 1);
18 **end**
19 **end**
20 Lines 13-25 Algorithm 4

- *RQ1 – How effective and efficient is our approach compared to the traditional delta debugging algorithm?* With this RQ, we aimed at answering whether the improved version of the traditional delta debugging algorithm performs better than the adapted version of the traditional delta debugging algorithm for stochastic CPSs. To this end, we compared both algorithms for both case study systems with the event-based test input reduction technique. We measured the effectiveness in terms of test input reduction ratio, failure reproduction ratio, and efficiency with execution time for both approaches.
- *RQ2 – How does the environment affect the performance of the delta debugging algorithm?* As shown in previous work [60], being aware of the environment improved significantly the efficiency of the delta debugging algorithm. However, the stochastic nature of the SUTs could affect the environment. For instance, for the Orona's case study system we take into account the static situations (i.e., when all the elevators are stopped with the doors opened), which could vary among the simulations of the same test case depending on the decisions of the dispatching algorithm. Therefore, with this RQ we aim at assessing whether for stochastic

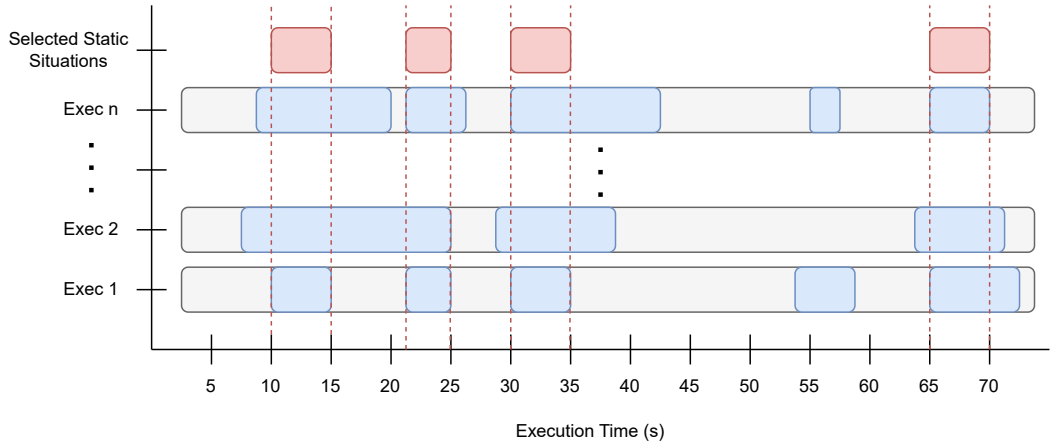


Fig. 4. Static States Selection example: In blue, static states for each execution of the original test input; in red dotted lines, convergence of static states across all executions; and in red, static states in common between all executions.

CPSs being aware of the environment also improves the performance of the delta debugging algorithm.

5.1 Test Input Characteristics

For the Orona's case study system, we used their Genetic Dispatching Algorithm and full-day real traffic data from 2 buildings. As test inputs, we considered 3 passenger profiles for each building, which are detailed in Table 2.

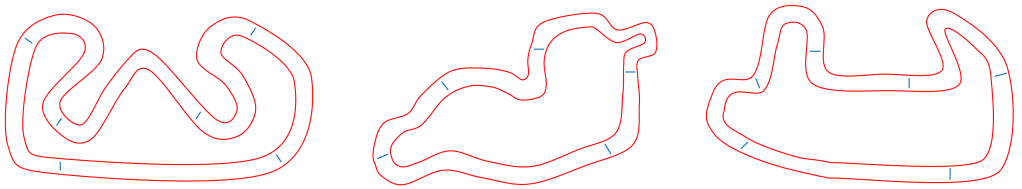
Table 2. Characteristics of the considered installations and test inputs for Orona's case study

Building	Test Input	# of elevators	# of floors	# of passengers	Execution Time in speed-up simulation (s)
Building 1	Test Input 1	3	12	3,769	368.88
	Test Input 2	3	12	3,105	352.10
	Test Input 3	3	12	3,294	361.22
Building 2	Test Input 4	6	10	6,558	351.87
	Test Input 5	6	10	5,452	250.73
	Test Input 6	6	10	4,467	223.82

For the Leo Rover case study system, as Figure 5 depicts, we designed 3 scenarios. These scenarios were manually built to replicate real-world F1 circuits, specifically Barcelona, Imola, and Las Vegas circuits. In each circuit, we added small blue lines to simulate events that caused the Leo Rover to stop for a certain period of time (e.g., a pedestrian or another vehicle crossing the road).

5.2 Execution Platform

On the one hand, we conducted the experiments for Orona's case study on a Windows 11 PC with a dual-core CPU Intel Core i5 7^{th} generation, and 16 GB RAM. As a simulator for executing the



(a) Barcelona Circuit (b) Imola Circuit (c) Las Vegas Circuit

Fig. 5. Realistic F1 circuit scenarios replicated for Leo Rover

tests, we used Elevate 8.19. On the other hand, for the Leo Rover case study, we used an Ubuntu virtual machine running on a Windows 11 PC. The virtual machine comprised 16GB RAM and an 12-core CPU AMD Ryzen 9 7900X with a 16GB NVIDIA GeForce RTX 4080 GPU.

5.3 Evaluation Metrics

We evaluated our approach from two perspectives: (1) efficiency and (2) effectiveness. The former relates to how fast the algorithm provides the minimized failure-inducing test input, whereas the latter refers to the quality of the provided minimized failure-inducing test inputs.

Efficiency: To assess the efficiency when comparing both approaches, we measured the execution time required by the algorithms to return the failure-inducing test input.

Effectiveness: Effectiveness relates to the quality of the results obtained by the different algorithms. We assessed the effectiveness of our approaches from two perspectives: (1) Test Input Reduction Ratio and (2) Failure Reproduction Rate. For the first one, we employed the Test Input Reduction Ratio with respect to the failing time ($TIRR_{ft}$), proposed in our previous work [59] which can be calculated as follows:

$$TIRR_{ft} = 1 - \frac{tet_{fail}(TI')}{tet_{fail}(TI)} \tag{1}$$

where, $tet_{fail}(TI')$ represents the test execution time required to trigger a failure using the minimized test input and $tet_{fail}(TI)$ represents the test execution time required to trigger a failure using the original test input. In addition, to complement the measure of the test input reduction ratio, we provide the simulation time needed to execute the failure for both case studies and the number of passengers of the minimized test input for Orona's case study. This way, we provide an intuitive measure of the effectiveness of our approach.

On the other hand, since the case studies are stochastic, it could be that sometimes the failure is not reproducible with the generated test case, so we measured the failure ratio ($TIFR$). This metric aims at giving confidence about the minimized test inputs to the user by assessing how reproducible the original failure is with the provided minimized test inputs. So, the higher this metric, the more probable it is to reproduce the failure with the minimized test inputs.

5.4 Experimental runs and statistical tests.

Due to the stochastic nature of the considered algorithms, we executed them multiple times. To answer the RQs, we executed each algorithm 10 times on each scenario. Therefore, in total for the Orona's case study system, we executed 2 (buildings) × 3 (test inputs) × 3 (traditional, improved DD and environment-aware improved DD) × 10 (runs) = 180 runs . On the other hand, for the Leo Rover case study system, we executed 3 (scenarios) × 3 (traditional, improved DD and environment-aware improved DD) × 10 (runs) = 90 runs. In summary, in our evaluation, we made a total of 270 runs,

which were parallelized by case study, each one on one computer. Notice that the multiple execution of the Leo Rover case study was not parallelized, since it required real-time simulation and the computational resources required to execute the experiments were high, so we could execute only one instance of it in the computer. For the Orona's case study, we were able to parallelize the executions by the algorithm and building, therefore, the cumulative number of executions for this case study was equivalent to 30 runs, since we could execute 6 instances of Elevate at the same time.

For all RQs, we assessed the statistical significance of the difference between the results by the different algorithms. We first analyzed how the data was distributed by employing the Shapiro-Wilk test. Since the data in some cases was normally distributed and in other cases it was not normally distributed, we used the ANOVA test and the Wilcoxon rank sum test according to the distribution of the data. We considered that there was statistical significance between the compared techniques when the p-value was below 0.05. In addition, we evaluated the effect sizes through the Vargha and Denaley's $\hat{A}_{12}$ value, which according to Romano et al. [50] the effect size of the $\hat{A}_{12}$ value can be categorized as *negligible* if $d < 0.147$, *small* if $d < 0.33$, *medium* if $d < 0.474$ and *large* if $d \geq 0.474$, where $d = 2|\hat{A}_{12} - 0.5|$.

5.5 Configuration of the algorithms

The configurability of our approach is essential, as it enables the adaptation from one case study to another. In addition to the typical modifications required to handle differences in the test inputs and simulation outputs, we had to adapt our approach in two perspectives: (1) the clustering process and (2) the comparison of a single run against the original test inputs.

As mentioned in Section 4.3.1, the first step of our approach is to categorize the failures in different clusters, therefore a bad clustering would drastically impact the effectiveness of our approach. Since clustering is an unsupervised method, selecting the optimal number of clusters based on the executions of the initial test input was essential. To select the number of clusters, we used the Bayesian Information Criterion (BIC) [53], which we found more suitable than Silhouette Score [51] and the Akaike Information Criterion (AIC) [1]. Although the Silhouette Score is widely used since it measures the cluster cohesion and separation, it does not penalize model complexity, making it less reliable for high-dimensional or noisy data, such as our data. Similarly, AIC evaluates how well a model fits but tends to favor more complex models, increasing the chances of overfitting. In contrast, BIC applies a stronger penalty for model complexity, preventing overfitting and providing a more robust and generalizable clustering solution.

There are several clustering methods that can be applicable when clustering a set of data. However, based on our clustering data, we used two different clustering methods, one for each case study system. For the Orona's case study system, we selected the K-Means clustering algorithm. K-Means is effective when the Euclidean distance metric provides a meaningful measure of similarity. In this case, it was meaningful to locate close passengers raising a failure, since the number of passengers is huge. In contrast, for the Leo Rover case study, we opted for the Gaussian Mixture (GMM) approach. This method assumes that the data is generated from a mixture of Gaussian distributions, making it well-suited for overlapping clusters. Unlike K-Means, which assigns each point to a single cluster, GMM provides soft assignments, allowing data points to belong to multiple clusters with varying probabilities. This property was particularly beneficial for capturing complex patterns and distributions. When trying the K-Means algorithm in this case study, as depicted in Figure 6, we found that it was very sensitive to small variations in the data. For instance, the best number of clusters was 3 for data with a standard deviation of 5 cm (i.e., almost negligible for this case study), whereas the optimal number of clusters for GMM was 1.

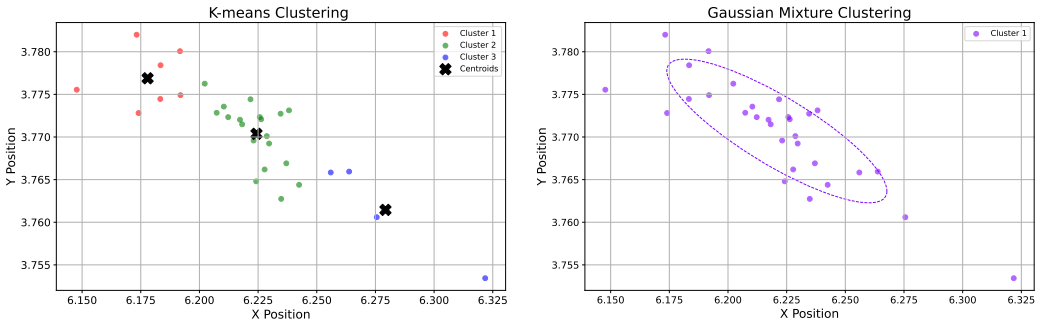


Fig. 6. K-Means clustering approach with 3 clusters as optimal cluster number VS Gaussian Mixture clustering approach with 1 cluster as optimal cluster number

6 ANALYSIS OF THE RESULTS

This section presents and discusses the results of our empirical evaluation according to the research questions introduced. For each research question, we first report the quantitative results and their statistical analysis, and then discuss the main observations and their implications. In particular, we examine the performance of the proposed techniques, their ability to reduce failure-inducing test inputs, and the extent to which minimization affects failure reproducibility under stochastic executions.

6.1 RQ1 - Performance

This research question aimed at assessing the effectiveness and efficiency of the proposed DD_{OS} compared to a conventional version of the Delta Debugging algorithm. To tackle this, we evaluated our approach against DD_S across two study systems. We evaluated the effectiveness using two evaluation metrics presented in Section 5.3: The Test Input Reduction Ratio (TIRR) and the Test Input Failure Reproduction Ratio (TIFR). Additionally, we assessed the efficiency of the algorithms by measuring the execution time.

6.1.1 Orona's case study. The execution time results (Figure 7) provide clear evidence of the improved efficiency achieved by the optimized approach (DD_{OS}) compared to DD_S. Across all scenarios, DD_{OS} consistently yields lower median execution times, indicating superior efficiency. In addition to the reduction in median values, the variability in execution time is also lower in five out of the six scenarios. Although statistical significance was observed in only two of the six scenarios, both the Vargha and Delaney Â12 and Cohen's *d* values indicate at least small effect sizes in favor of *DDOS* in all cases (Table 3). More specifically, Scenarios 5 and 6 show statistically significant improvements in favor of DD_{OS} with large effect sizes. A post-hoc analysis of the original failing test inputs revealed that, in all scenarios where DD_{OS} outperforms DD_S, the failure-inducing behavior occurs near the end of the test input. In these cases, reproducing the failure with the original test input required at least 200 seconds. In contrast, for the remaining test cases, failure reproduction required between 100 and 175 seconds. These findings suggest that the execution time improvement provided by DD_{OS} becomes more pronounced as the time required to reproduce failures increases, making the optimized version of the Delta Debugging algorithm particularly suitable for long-running test cases.

Figure 8 presents the comparison of the Test Input Reduction Ratio (TIRR) between both approaches. The results show that the optimized version consistently achieves higher median reduction

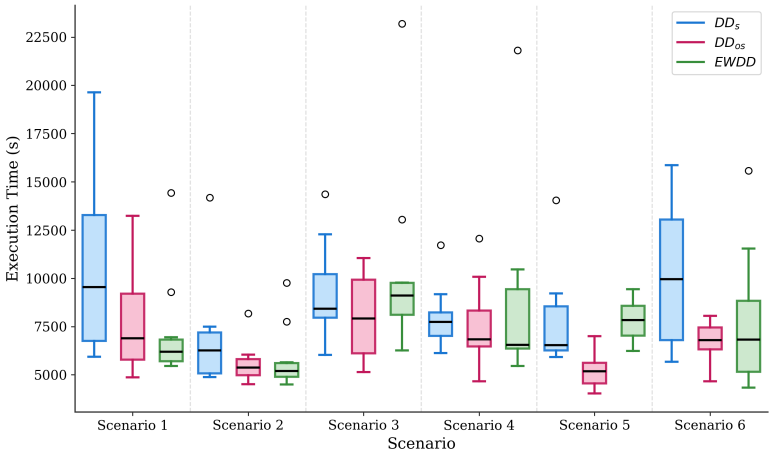


Fig. 7. Execution time for Orona's case study

ratios across all scenarios, indicating a greater capability to reduce test inputs. Furthermore, similarly to the execution time results, the distribution across different runs is tighter compared to the baseline approach. As shown in Table 3, statistically significant differences in favor of DD_{OS} were observed in three scenarios (Scenarios 1, 5, and 6), all with large effect sizes. In the remaining scenarios, although no statistical significance was detected, the observed effect sizes still favor DD_{OS}, suggesting that this approach is generally more effective at reducing test inputs than the baseline algorithm. When the failure point occurs near the beginning of the test input, the initial reduction step in both algorithms substantially decreases the input size, leaving limited room for further reduction. Consequently, both approaches exhibit similar performance for scarcely reducible test inputs. However, for longer test inputs, the increased flexibility of DD_{OS} becomes advantageous. Its more permissive oracle, which compares statistically single-run outcomes against multiple executions, allows it to achieve greater reductions than DD_S, highlighting its superior adaptability in handling more complex scenarios.

Figure 9 reports the Test Input Failure Reproduction Ratio (TIFR), which measures the ability of the reduced test inputs to still trigger the original failure. For Scenarios 2, 4, and 5, both algorithms achieved identical results, reaching the maximum reproduction ratio (100%). However, in the remaining three scenarios, DD_{OS} obtained better results. As shown in Table 3, Scenario 1 exhibits statistically significant differences in favor of DD_{OS} with a large effect size. In the other scenarios, although no statistical significance was observed, the effect sizes consistently favor the optimized approach. Furthermore, when comparing both approaches with the original test input, the minimized test inputs achieved higher failure reproduction ratios in all cases. This suggests that the reduced test inputs increase the likelihood of reproducing the original failure compared to the original test inputs.

6.1.2 Leo Rover case study. As in the case of Orona's case study system, within the LeoRover case study system, the median test execution time values were also lower for DD_{OS} compared to DD_S, as depicted in Figure 10. Across all scenarios, DD_{OS} consistently yielded lower median execution times, indicating superior efficiency. However, in two of the circuits (Circuits 2 and 3), the variability of DD_{OS} was higher. For these two circuits, there was no statistical significance, although the Â12 were in favor of DD_{OS} (Table 4).

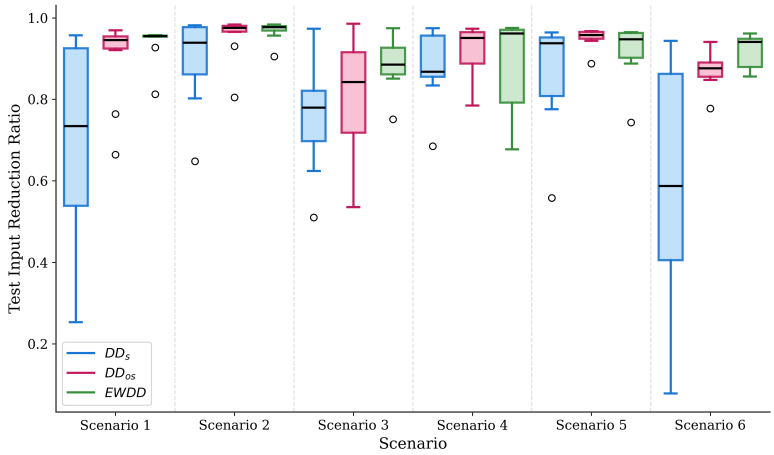


Fig. 8. Test Input Reduction Ratio for Orona's case study

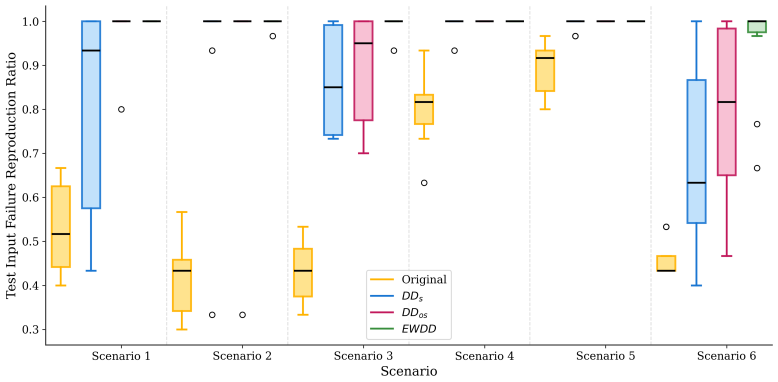


Fig. 9. Test Input Failure Reproduction for Orona's case study

Figure 11 presents the comparison of the Test Input Reduction Ratio (TIRR) between both approaches for the Leo Rover case study system. The results show that the DD_{OS} algorithm consistently achieves higher reduction of the test inputs, indicating a greater capability to reduce test inputs, which leads to lower test execution times, and therefore, improving debugging efficiency. These results were statistically significant, with large effect sizes for Circuits 1 and 2, but there was no statistical significance in Circuit 3, as shown in Table 4.

Figure 12 reports the Test Input Failure Reproduction Ratio (TIFR), which measures the ability of the reduced test inputs to trigger the original failure, thereby helping engineers debug their faults. In this case study system, both delta debugging algorithms significantly help in reproducing the original failure. Indeed, for Circuits 1 and 2, the median TIFR value for the DD_{OS} algorithm was 100%, indicating that in most cases this algorithm was able to reproduce the exact same failure as the original one. In Circuit 3, this was slightly reduced, although results were good too (median values above 80%). The results were slightly in favor of the DD_{OS} in Circuits 1 and 2, and comparable with DD_S in Circuit 3.

Table 3. Statistical test for efficiency and effectiveness comparison DD_{OS} vs DD_S and $EWDD_{OS}$ vs DD_{OS} for Orona's case stuyd

	Scenario	Execution time			TIRR			TIFR		
		$\hat{A}_{12}$	p_value	Cohen's d	$\hat{A}_{12}$	p_value	Cohen's d	$\hat{A}_{12}$	p_value	Cohen's d
DD_{OS} vs DD_S	Scenario 1	0.27	0.0917	-0.79	0.81	0.0190	1.05	0.76	0.0211	1.04
	Scenario 2	0.36	0.2899	-0.60	0.69	0.1400	0.67	0.54	0.5838	0.03
	Scenario 3	0.38	0.2994	-0.47	0.64	0.4566	0.34	0.55	0.6968	0.27
	Scenario 4	0.41	0.5768	-0.25	0.62	0.3643	0.54	0.55	0.3173	0.44
	Scenario 5	0.08	0.0014	-1.31	0.79	0.0283	0.84	0.60	0.1462	0.67
	Scenario 6	0.26	0.0131	-1.23	0.77	0.0083	1.32	0.66	0.2428	0.54
$EWDD_{OS}$ vs DD_{OS}	Scenario 1	0.44	0.6501	-0.13	0.66	0.2259	0.47	0.55	0.3173	0.44
	Scenario 2	0.44	0.6501	0.16	0.5	1.0000	0.32	0.50	0.9421	0.42
	Scenario 3	0.64	0.2899	0.59	0.65	0.1732	0.63	0.76	0.0186	1.07
	Scenario 4	0.51	0.9397	0.35	0.52	0.8797	-0.34	0.50	1.0000	0.00
	Scenario 5	0.97	<0.0001	2.66	0.32	0.1857	-0.57	0.50	1.0000	0.00
	Scenario 6	0.51	0.4041	0.38	0.78	0.0342	1.14	0.74	0.0524	0.90

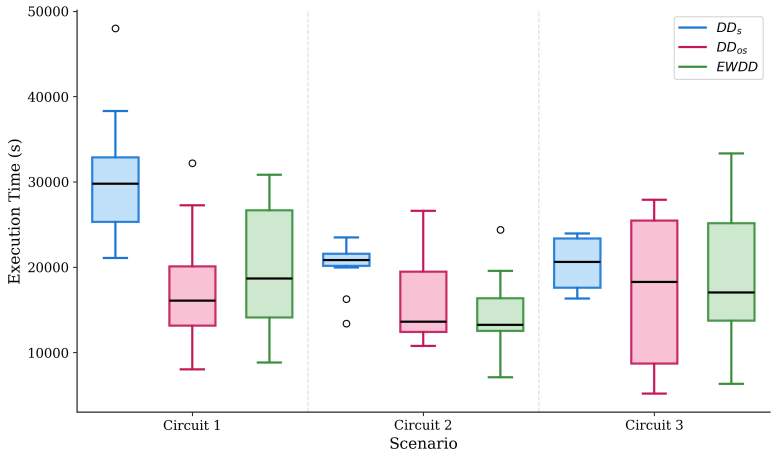


Fig. 10. Execution time for Leo Rover's case study

6.1.3 Summary of the results and RQ1 answer. In summary, the results from both Orona's industrial case study and the Leo Rover open-source case study demonstrate that the optimized Delta Debugging algorithm (DD_{OS}) outperforms the stochastic baseline (DD_S) in terms of efficiency and effectiveness. In Orona's scenarios, DD_{OS} consistently achieves lower median execution times, higher test input reduction ratios (TIRR), and improved or equivalent failure reproduction ratios (TIFR), with statistical significance and large effect sizes in several cases, particularly for longer-running tests where failures occur later. Similarly, for the Leo Rover circuits, DD_{OS} yields reduced execution times, superior TIRR values (statistically significant in two circuits), and high TIFR medians (often 100%), indicating reliable failure reproduction despite increased variability in some scenarios. Overall, these findings underscore DD_{OS}'s adaptability to stochastic environments, making it more suitable for debugging CPSs on flaky simulators.

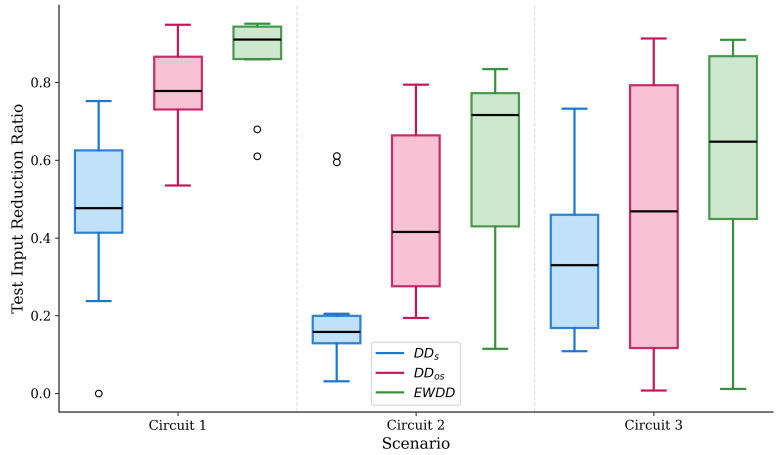


Fig. 11. Test Input Reduction Ratio for Leo Rover's case study

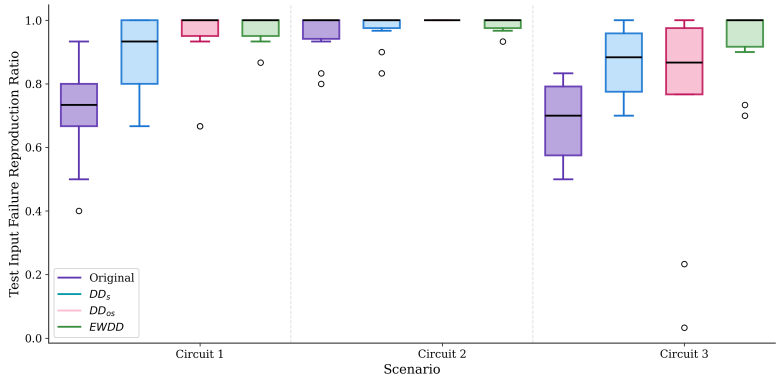


Fig. 12. Test Input Failure Reproduction for Leo Rover's case study

Table 4. Statistical test for efficiency and effectiveness comparison DD_{OS} vs DD_S and $EWDD_{OS}$ vs DD_{OS} for Leo Rover's case study

	Scenario	Execution time			TIRR			TIFR		
		$\hat{A}_{12}$	p_value	Cohen's d	$\hat{A}_{12}$	p_value	Cohen's d	$\hat{A}_{12}$	p_value	Cohen's d
DD_{OS}	Circuit 1	0.11	0.0015	-1.66	0.92	0.0020	1.61	0.58	0.4672	0.31
vs	Circuit 2	0.30	0.1305	-0.84	0.85	0.0081	1.04	0.65	0.0681	0.73
DD_S	Circuit 3	0.44	0.3181	-0.45	0.56	0.4211	0.36	0.46	0.7892	-0.49
$EWDD_{OS}$	Circuit 1	0.57	0.5767	0.25	0.74	0.0696	0.73	0.52	0.8152	0.45
vs	Circuit 2	0.44	0.6501	-0.30	0.69	0.2022	0.59	0.35	0.0678	-0.80
DD_{OS}	Circuit 3	0.56	0.7399	0.15	0.62	0.3903	0.39	0.68	0.1528	0.74

RQ1: The proposed approach (DD_{OS}) is more effective and efficient than the traditional Delta Debugging algorithm (DD_S), as it consistently delivers lower execution times, higher test input reduction ratios, and comparable or superior failure reproduction rates across both case studies, with statistical significance observed in multiple scenarios.

6.2 RQ2 - Effect of the environment

RQ2 investigates whether incorporating environment-awareness into the delta debugging process improves its performance when dealing with stochastic CPSs. In particular, we compare the DD_{OS} algorithm with its environment-wise extension ($EWDD_{OS}$), which leverages static situations to guide the minimization process. The analysis focuses on effectiveness (quality of the minimized test inputs) and efficiency (execution time).

6.2.1 Orona's case study. For the industrial case study, incorporating environment-awareness has a clear and consistent impact on performance.

Regarding TIRR, $EWDD_{OS}$ achieved similar reduction ratios comparable to DD_{OS}, where there was no statistical significance (Table 3). However, in Scenario 6, $EWDD_{OS}$ improved DD_{OS} with statistical significance. The improvement is mainly observed in a test input where well-defined static situations exist (i.e., states in which all elevators are stopped with doors closed). These static configurations act as stable synchronization points across executions, despite the stochastic nature of the genetic dispatching algorithm. By aligning reductions with these shared environmental states, $EWDD_{OS}$ avoids removing critical context required to reproduce the failure, while still enabling aggressive minimization. As a result, the minimized test inputs preserve the original failure characteristics and maintain high failure reproduction rates.

In terms of TIFR, $EWDD_{OS}$ demonstrates slightly more stable behavior across repeated executions. In five out of six scenarios, DD_{OS} already achieved nearly perfect reproduction ratio. However, Scenario 6 was not the case, for which $EWDD_{OS}$ clearly outperformed, with statistical significance DD_{os}. Because the algorithm reduces the test input at meaningful environmental boundaries, the resulting minimized inputs tend to reproduce the selected failure cluster more consistently. This effect is particularly visible in long full-day simulations, where purely sequence-based reductions (as in DD_{OS}) may occasionally isolate statistically unstable intermediate states.

In terms of Execution Time, in 5 out of 6 scenarios, there was no statistical significance. However, in Scenario 5, DD_{OS} outperformed $EWDD_{OS}$ with statistical significance. This meant that, on average, DD_{OS} spent 2607.39 seconds less than $EWDD_{OS}$, for which the environment was not helpful in this particular environment.

6.2.2 Leo Rover case study. In contrast to the industrial case study, incorporating environment-awareness has a more limited and mixed impact on performance for the Leo Rover open-source case study system. Regarding execution time, $EWDD_{OS}$ and DD_{OS} exhibit comparable median execution times across all three circuits, with no statistical significance detected in any case (Table 4). The $\hat{A}_{12}$ values hover around 0.5 and Cohen's d effect sizes remain small or negligible, indicating that the overhead of identifying and aligning static situations neither reduces nor increases minimization time in this setting. This outcome is attributable to the continuous navigation dynamics of the rover, where pronounced static synchronization points (e.g., fixed positions or headings) are less frequent or consistent across runs because of the combined randomness from the Gazebo physics engine, image-processing delays, and ROS communication.

For the Test Input Reduction Ratio (TIRR), $EWDD_{OS}$ achieves slightly higher median reduction ratios than DD_{OS} in Circuits 1 and 2 (medium effect sizes), although statistical significance is not reached ($p > 0.06$). In Circuit 3, the performances are essentially equivalent. By pruning waypoint sequences up to shared environmental states before the failure, the environment-aware variant enables modestly more aggressive minimization without losing the failure-inducing context; however, the benefit is smaller than in the elevator case, where discrete static states (elevators stopped with doors open) are more reliably identifiable.

In terms of the Test Input Failure Reproduction Ratio (TIFR), results are again mixed and without statistical significance. $EWDD_{OS}$ slightly improves reproduction in Circuit 3, while DD_{OS} performs marginally better in Circuit 2 (p = 0.0678, close but still non-significant). Overall, both approaches maintain high reproduction rates (medians often above 80% and frequently reaching 100% as already observed for DDOS in RQ1), confirming that the additional environmental alignment does not substantially enhance reliability in a system affected by multiple independent randomness sources.

Taken together, the environment-aware extension provides only modest or neutral gains for the Leo Rover, highlighting that the value of static-situation guidance is case-dependent and less pronounced when failures arise from continuous, image-driven control loops rather than discrete event sequences.

6.2.3 Summary of the results and RQ2 answer. In summary, the impact of incorporating environment-awareness into the delta debugging process is clearly *context-dependent*. In the Orona industrial case study, $EWDD_{OS}$ provides tangible benefits over DD_{OS}, particularly in scenarios where well-defined static situations exist and can be reliably identified across executions. In these cases, aligning reductions with meaningful environmental synchronization points improves stability and, in specific scenarios (notably Scenario 6), leads to statistically significant improvements in TIRR and TIFR. The approach preserves the failure-inducing context more robustly in long-running simulations, where purely sequence-based minimization may isolate statistically unstable states. Although execution time improvements are not systematic, and in one scenario ($S5$) DD_{OS} remains faster, the effectiveness gains in critical scenarios demonstrate the practical value of environment-awareness in structured CPSs with discrete, identifiable states.

In contrast, for the Leo Rover case study, the benefits of $EWDD_{OS}$ are modest and statistically non-significant. The rover operates in a continuous control loop influenced by multiple independent sources of randomness (physics simulation, perception noise, communication delays), where clear and stable environmental synchronization points are less frequent. As a result, environment-guided reductions neither substantially improve test input reduction nor consistently enhance failure reproduction. Both DD_{OS} and $EWDD_{OS}$ achieve comparable execution times and high reproduction ratios, indicating that environment-awareness does not introduce harm but also does not provide clear advantages in this setting.

Overall, the results indicate that environment-aware delta debugging is particularly beneficial in CPSs characterized by discrete, well-defined environmental states that can serve as reliable synchronization anchors across stochastic executions. Its effectiveness diminishes in systems dominated by continuous dynamics and perception-driven variability, where static situations are less stable or harder to exploit.

RQ2: Incorporating environment-awareness into delta debugging improves performance when the CPS exhibits identifiable and repeatable static environmental states, as demonstrated in the Orona case study. However, in systems governed by continuous dynamics and multiple stochastic sources, such as the Leo Rover, the benefits are limited. Thus, the effectiveness of environment-aware delta debugging is strongly dependent on the structural properties of the environment.

7 DISCUSSION

This section presents the lessons learned from our empirical evaluation and discusses the main threats to validity and the measures adopted to mitigate them.

7.1 Lessons learned

Our empirical evaluation provides several insights into the application of delta debugging to stochastic Cyber-Physical Systems (CPSs), extending beyond the comparative performance of the proposed algorithms.

Lesson 1 – Failure-inducing input reduction minimizes test flakiness: The most notable observation is that minimizing failure-inducing test inputs increased for all the cases the Test Input Failure Reproduction Ratio (TIFR) with respect to the original executions. This result indicates that test-input reduction does not merely preserve failure-inducing conditions but can actively improve their reproducibility. We hypothesize that this behavior arises because long CPS executions accumulate multiple sources of stochastic variability (e.g., simulator artifacts, scheduling effects, communication delays, perception noise, and randomized algorithmic decisions) that are unrelated to the root cause of the failure. Each additional execution step therefore introduces further opportunities for behavioral divergence across repeated runs. By removing execution segments that are not causally required to trigger the failure, delta debugging reduces the exposure of the system to these stochastic influences, effectively narrowing the execution to the critical conditions responsible for the fault. Consequently, the minimized test inputs are less susceptible to incidental randomness and reproduce the original failure more consistently. This finding suggests that test-input minimization can simultaneously support fault localization and mitigate execution flakiness, representing an additional benefit that has received limited attention in previous work. Moreover, one could argue that test input minimization simultaneously could improve fault localization by reducing execution length and increases the reproducibility of the remaining failure, thereby lowering the overall cost of debugging stochastic CPSs.

Lesson 2 – Reliable debugging of stochastic CPSs requires statistical reasoning rather than deterministic failure preservation: Classical delta debugging assumes that the outcome of a reduced test input can be determined from a single execution. Our results demonstrate that this assumption is generally invalid in stochastic CPSs. Instead, determining whether a reduction preserves the original failure requires repeated executions combined with statistical analyses capable of distinguishing genuine behavioral equivalence from execution variability. Our results indicate that reliable debugging approaches for CPSs tested under flakiness require statistical reasoning rather than deterministic pass/fail decisions.

Lesson 3 – The degree of execution variability determines the most suitable minimization strategy: The experiments indicate that the relative performance of the proposed algorithms depends on the severity of execution flakiness. Under moderate variability, the speculative search strategy employed by DD_{OS} substantially reduces debugging time while maintaining comparable reduction quality. Conversely, when execution variability becomes more pronounced, the conservative validation strategy adopted by DDS provides greater robustness by verifying each candidate reduction through repeated statistical evaluation. These results suggest that debugging algorithms should adapt their validation strategy according to the stochastic characteristics of the target system rather than relying on a single fixed reduction policy.

Lesson 4 – Longer executions provide greater opportunities for effective reduction: The effectiveness of delta debugging is closely related to the position of the failure within the original execution. Failures occurring later in long executions provide greater opportunities for removing behavior that is unrelated to the observed fault, resulting in both larger reduction ratios and greater reductions in debugging time. Consequently, the relative advantages of optimized reduction strategies become increasingly significant as the temporal distance between the beginning of the execution and the failure increases.

Lesson 5 – Failure characterization should be adapted to the properties of the application domain: The statistical comparison of failures depends critically on the quality of the failure clustering process. Our evaluation shows that no single clustering technique is universally appropriate. K-Means produced reliable failure groupings for the discrete and high-volume passenger data generated by the elevator dispatching system, whereas Gaussian Mixture Models more accurately represented the continuous and overlapping failure distributions observed in the autonomous robot. Similarly, the Bayesian Information Criterion (BIC) consistently provided more reliable model selection than alternative criteria in our experiments. These observations highlight that the characterization of stochastic failures should reflect the statistical properties of the underlying system rather than relying on generic clustering methods.

7.2 Threats to Validity

We now discuss the main threats to the validity of our empirical evaluation and the measures taken to mitigate them.

Internal validity: A potential internal validity threat concerns the configuration of the proposed algorithms. In particular, the results may be influenced by the number of repeated executions used to statistically validate a candidate test input, the confidence thresholds employed by the statistical tests, and the criteria used to accept or reject a reduction. We selected these values based on established statistical conventions and preliminary experimentation, and applied the same configuration consistently across the compared algorithms, making the comparison fair. Nevertheless, alternative configurations may lead to different trade-offs between execution cost and confidence in the preservation of the failure. Another threat concerns the implementation of the three algorithms. To reduce this risk, the approaches share the same execution, failure-analysis, and test input manipulation infrastructure, differing only in the reduction and validation strategies under comparison.

Construct validity: Determining whether two executions exhibit the same failure relies on application-specific failure representations and clustering strategies. Incorrectly grouped or separated failures could affect the statistical validation of candidate reductions. We mitigated this threat by selecting clustering techniques according to the statistical properties of each case study and by evaluating their suitability using established model-selection criteria.

Conclusion validity: The stochastic behavior of both case study systems may introduce random variation into the observed reduction ratios, failure reproduction ratios, and execution times. We mitigated this threat by executing the algorithms repeatedly and analyzing the resulting distributions using statistical significance tests and effect-size measures. Nevertheless, the high computational cost of CPS simulations limits the number of repetitions that can be performed. Moreover, the absence of statistical significance in some comparisons does not necessarily demonstrate equivalence between the algorithms, particularly when the observed differences are consistent but the number of available scenarios is limited. Such results should therefore be interpreted as insufficient evidence of a difference rather than evidence that the approaches perform identically.

External validity: The generalizability of our findings is limited by the number of systems and failure scenarios considered. Our evaluation includes two complementary CPSs: an industrial elevator-dispatching system whose stochasticity originates from a randomized optimization algorithm, and an autonomous mobile robot whose variability arises primarily from the simulation and execution infrastructure. This diversity allows us to study distinct sources of stochasticity. Furthermore, the industrial case study strengthens the practical relevance of the evaluation by demonstrating that the proposed techniques scale and can be integrated into a real development and testing process. Nevertheless, additional CPS domains, simulators, controllers, and failure

types are required before broadly generalizing the results. The applicability of the environment-aware approach is further constrained to systems that expose stable operational states from which simulations can be reliably initialized.

8 RELATED WORK

Delta Debugging originated with Hildebrandt and Zeller's early work on simplifying failure-inducing inputs and was fully established by the *ddmin* algorithm in the seminal article by Zeller and Hildebrandt, which framed reduction as a divide-and-conquer search over subsets and complements of a failing input [36, 65]. In its classical form, Delta Debugging guarantees a *1-minimal* result, i.e., a failure-inducing input from which no single remaining unit can be removed without losing the target property [65]. A rich line of follow-up work has specialized DD to particular input structures and debugging settings. HDD leveraged hierarchical structure to reduce tree-based inputs such as XML more efficiently and often more aggressively than flat Delta Debugging [47]. Iterative Delta Debugging extended the idea to settings with masking or cascading faults, repeatedly applying Delta Debugging until an originally hidden bug can be isolated [11]. ProbDD replaced fixed partitioning with probabilistic reasoning about element relevance and demonstrated substantial improvements over existing ddmin-based reducers such as HDD and CHISEL [63]. Delta Debugging has also been instantiated in domain-specific debugging workflows, for example for shrinking SMT formulas [23] and for reducing failure-inducing circumstances in microservice systems [66]. In CPSs, our prior work adapted Delta Debugging to long operational traces and further introduced an environment-aware variant that exploits stable states of the simulated environment to accelerate reduction [59, 60]. However, the common assumption across this literature is that each candidate reduction can be validated from a deterministic outcome. Our work departs from this assumption by extending Delta Debugging to stochastic CPSs, replacing deterministic failure preservation with repeated execution and statistical validation, and by introducing optimized and environment-aware stochastic variants that make such validation practical under flaky simulation.

The broader CPS literature has primarily focused on *finding* failures rather than *minimizing* them once detected. Simulation-based test generation and prioritization for CPS models aim at efficiently exploring long-running dynamic behaviors under high execution cost [7, 45, 48]. Likewise, falsification research searches for counterexamples to temporal-logic requirements and has evolved into a mature benchmark- and tool-centric ecosystem [46]. More directly related to debugging, approaches such as CPSDebug explain how failures propagate through Simulink/Stateflow models by combining testing, specification mining, and failure analysis [18]. Ghazal et al [33] propose a counterfactual explanation approach along with assertion inference approach to debug CPS failures by helping them interpret under what circumstances the systems fail. These contributions are highly relevant because they highlight why CPS debugging is difficult: failures often emerge from long, heterogeneous interactions among software, controllers, timing, and physical dynamics. However, they do not address the specific problem considered in this paper, namely reducing already failing CPS test inputs under stochastic executions. In contrast to prior work on test generation, falsification, monitoring, or gray-box failure explanation, our work focuses on *statistical test input reduction* after failure discovery, and explicitly targets both algorithm-induced and infrastructure-induced stochasticity.

Recent empirical evidence indicates that simulator flakiness is a common and practically significant problem for autonomous and robotic CPSs. Amini et al. [3] found that repeated executions of the same autonomous driving test can lead to both quantitative and qualitative inconsistencies, including changes in pass/fail verdicts, and that this variability materially affects the behavior of randomized testing algorithms. This observation is consistent with systems research on robotic middleware, which shows that ROS 2 callback execution is inherently nondeterministic under the

default publish-subscribe execution model, with further variability caused by message interleavings and network latency in distributed deployments [52]. Additional execution variability may arise from physics simulation, image acquisition, DNN inference latency, and accelerator contention in perception-control pipelines [31, 32]. Existing work has mainly focused on predicting flaky tests, enforcing deterministic middleware execution, or improving simulator trustworthiness. Our work is complementary to these. Instead of assuming away flakiness, we apply a novel delta debugging algorithm that remain sound under stochastic execution by combining repeated reruns, failure clustering, and statistical validation. In this sense, our techniques do not compete with flaky test prediction or deterministic replay. Instead, they provide a debugging method that remains applicable when stochastic execution variability cannot be fully eliminated.

9 CONCLUSION AND FUTURE WORK

Flaky executions complicate the debugging of CPSs because identical test inputs may produce inconsistent outcomes, preventing conventional Delta Debugging from reliably preserving the original failure. To address this problem, we proposed stochastic, optimized, and environment-aware Delta Debugging techniques that combine statistical validation with speculative reduction and rollback. We evaluated the approaches on an autonomous mobile robot and an industrial elevator-dispatching system. The results show that the proposed techniques substantially reduce failure-inducing test inputs while maintaining their failure behavior and decreasing the algorithm's execution time. Most notably, the minimized inputs frequently achieved higher failure reproduction ratios than the original executions. This suggests that removing non-essential portions of a test input reduces the system's exposure to incidental sources of stochastic variability, thereby making the target failure less flaky and easier to reproduce. Overall, the results indicate that Delta Debugging can both simplify failure-inducing executions and improve their reproducibility. Future work will evaluate this relationship across additional CPS domains and develop adaptive validation strategies based on the observed degree of flakiness.

ACKNOWLEDGMENTS

Pablo Valle and Aitor Arrieta are part of the Systems and Software Engineering research group of Mondragon Unibertsitatea (IT1919-26), supported by the Department of Education, Universities and Research of the Basque Country. Aitor Arrieta is supported by the Spanish Ministry of Science, Innovation and Universities (project PID2023-152979OA-I00), funded by MCIU /AEI /10.13039/501100011033 / FEDER, UE. Pablo Valle is supported by the Pre-doctoral Program for the Formation of Non-Doctoral Research Staff of the Education Department of the Basque Government (Grant n. PRE_2025_2_0252). S. Ali is supported by the Co-tester project (Project# 314544), funded by the Research Council of Norway.

REFERENCES

[1] Hirotugu Akaike. 1974. A new look at the statistical model identification. *IEEE transactions on automatic control* 19, 6 (1974), 716–723.
[2] Rajeev Alur. 2015. *Principles of cyber-physical systems.* MIT press.
[3] Mohammad Hossein Amini, Shervin Naseri, and Shiva Nejati. 2024. Evaluating the impact of flaky simulators on testing autonomous driving systems. *Empirical Software Engineering* 29, 2 (2024), 47.
[4] Aitor Arrieta, Joseba Andoni Agirre, and Goiuria Sagardui. 2020. Seeding strategies for multi-objective test case selection: An application on simulation-based testing. In *Proceedings of the 2020 genetic and evolutionary computation conference*. 1222–1231.
[5] Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. 2017. Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal* 25, 3 (2017), 1041–1083.

[6] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. 2019. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology* 114 (2019), 137–154.

[7] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2017. Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems. *IEEE Transactions on Industrial Informatics* 14, 3 (2017), 1055–1066.

[8] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2017. Search-based test case generation for cyber-physical systems. In *2017 IEEE congress on evolutionary computation (CEC)*. IEEE, 688–697.

[9] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. 2016. Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 1053–1060.

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

[11] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13 (2011), 223–246.

[12] Srikesh G Arunajadai, Scott J Uder, Robert B Stone, and Irem Y Tumer. 2004. Failure mode identification through clustering analysis. *Quality and Reliability Engineering International* 20, 5 (2004), 511–526.

[13] Jon Ayerdi, Aitor Garciandia, Aitor Arrieta, Wasif Afzal, Eduard Enoiu, Aitor Agirre, Goiuria Sagardui, Maite Arratibel, and Ola Sellin. 2020. Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE, 280–290.

[14] Jon Ayerdi, Asier Iriarte, Pablo Valle, Ibai Roman, Miren Illarramendi, and Aitor Arrieta. 2024. Marmot: Metamorphic runtime monitoring of autonomous driving systems. *ACM Transactions on Software Engineering and Methodology* 34, 1 (2024), 1–35.

[15] Jon Ayerdi, Sergio Segura, Aitor Arrieta, Goiuria Sagardui, and Maite Arratibel. 2020. Qos-aware metamorphic testing: An elevation case study. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–114.

[16] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1264–1274.

[17] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *The impact of control technology* 12, 1 (2011), 161–166.

[18] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. 2021. CPSDebug: Automatic failure explanation in CPS models. *International Journal on Software Tools for Technology Transfer* (2021), 1–14.

[19] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. 2023. Property-based mutation testing. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 222–233.

[20] Maite Beamurgia, Rosa Basagoiti, I Rodríguez, and V Rodriguez. 2016. A modified genetic algorithm applied to the elevator dispatching problem. *Soft Computing* 20 (2016), 3595–3609.

[21] Christian Birchler, Cyrill Rohrbach, Hyeongkyun Kim, Alessio Gambi, Tianhai Liu, Jens Horneber, Timo Kehrer, and Sebastiano Panichella. 2023. TEASER: Simulation-based CAN Bus Regression Testing for Self-driving Cars Software. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2058–2061.

[22] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. 2016. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th international conference on software engineering companion*. 789–792.

[23] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 1–5.

[24] Henian Chen, Patricia Cohen, and Sophie Chen. 2010. How big is a big odds ratio? Interpreting the magnitudes of odds ratios in epidemiological studies. *Communications in Statistics—simulation and Computation®* 39, 4 (2010), 860–864.

[25] Oscar Cornejo, Fabrizio Pastore, and Lionel C Briand. 2021. Mutation analysis for cyber-physical systems: Scalable solutions and results in the space domain. *IEEE Transactions on Software Engineering* 48, 10 (2021), 3913–3939.

[26] Anthony Corso, Robert Moss, Mark Koren, Ritchie Lee, and Mykel Kochenderfer. 2021. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of Artificial Intelligence Research* 72 (2021), 377–428.

[27] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. 2011. Modeling cyber–physical systems. *Proc. IEEE* 100, 1 (2011), 13–28.

[28] William Dickinson, David Leon, and A Fodgurski. 2001. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* IEEE, 339–348.

[29] Nicholas DiGiuseppe and James A Jones. 2012. Concept-based failure clustering. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*. 1–4.

[30] John C Eidson, Edward A Lee, Slobodan Matic, Sanjit A Seshia, and Jia Zou. 2011. Distributed real-time software for cyber–physical systems. *Proc. IEEE* 100, 1 (2011), 45–59.

[31] Daniel Enright, Yecheng Xiang, Hyunjong Choi, and Hyoseung Kim. 2024. Paam: A framework for coordinated and priority-driven accelerator management in ros 2. *arXiv preprint arXiv:2404.06452* (2024).

[32] Diego Ferigo, Silvio Traversaro, Giorgio Metta, and Daniele Pucci. 2020. Gym-ignition: Reproducible robotic simulations for reinforcement learning. In *2020 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 885–890.

[33] Zaid Ghazal, Hadiza Yusuf, and Khouloud Gaaloul. 2026. Towards Counterfactual Explanation and Assertion Inference for CPS Debugging. In *2026 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 623–634.

[34] Liping Han, Shaukat Ali, Tao Yue, Aitor Arrieta, and Maite Arratibel. 2022. Uncertainty-aware Robustness Assessment of Industrial Elevator Systems. *ACM Transactions on Software Engineering and Methodology* (2022).

[35] Liping Han, Tao Yue, Shaukat Ali, Aitor Arrieta, and Maite Arratibel. 2022. Are elevator software robust against uncertainties? results and experiences from an industrial case study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1331–1342.

[36] Ralf Hildebrandt and Andreas Zeller. 2000. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. 135–145.

[37] Yunfei Hou, Yunjie Zhao, Aditya Wagh, Longfei Zhang, Chunming Qiao, Kevin F Hulme, Changxu Wu, Adel W Sadek, and Xuejie Liu. 2015. Simulation-based testing and evaluation tools for transportation cyber–physical systems. *IEEE Transactions on Vehicular Technology* 65, 3 (2015), 1098–1108.

[38] Aaron Kane, Thomas Fuhrman, and Philip Koopman. 2014. Monitor based oracles for cyber-physical system testing: Practical experience report. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 148–155.

[39] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. 2024. Simulation-based testing of unmanned aerial vehicles with aerialist. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 134–138.

[40] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. Ieee, 2149–2154.

[41] Jaekwon Lee, Fabrizio Pastore, and Lionel Briand. 2026. Fuzzing-based mutation testing of C/C++ software in cyber-physical systems. *Empirical Software Engineering* 31, 1 (2026), 20.

[42] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*. IEEE, 60–69.

[43] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[44] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*. 595–606.

[45] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2018. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering* 45, 9 (2018), 919–944.

[46] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. 2020. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 372–384.

[47] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[48] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C Briand, Stephen Foster, and David Wolfe. 2019. Evaluating model testing and model checking for finding requirements violations in Simulink models. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1015–1025.

[49] Olek Osikowicz, Phil McMinn, and Donghwan Shin. 2024. Empirically Evaluating Flaky Tests for Autonomous Driving Systems in Simulated Environments. In *2025 IEEE/ACM International Flaky Tests Workshop (FTW) Proceedings*. Institute of Electrical and Electronics Engineers (IEEE).

[50] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*. Citeseer, 1–51.

[51] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[52] Simon Sagmeister, Marcel Weinmann, Phillip Pitschi, and Markus Lienkamp. 2026. RSLCPP-Deterministic Simulations Using ROS 2. *arXiv preprint arXiv:2601.07052* (2026).
[53] Gideon Schwarz. 1978. Estimating the dimension of a model. *The annals of statistics* (1978), 461–464.
[54] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 359–371.
[55] Christopher S Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wąsowski. 2024. ROBUST: 221 bugs in the Robot Operating System. *Empirical Software Engineering* 29, 3 (2024), 57.
[56] Graham JG Upton. 1992. Fisher's exact test. *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 155, 3 (1992), 395–402.
[57] Pablo Valle, Shaukat Ali, and Aitor Arrieta. 2026. *Github Repository of "Delta Debugging for Cyber- Physical Systems with Flaky Test Executions"*. https://github.com/pablovalle/DeltaDebugging4CPSs GitHub repository.
[58] Pablo Valle, Shaukat Ali, and Aitor Arrieta. 2026. *Replication Package of "Delta Debugging for Cyber- Physical Systems with Flaky Test Executions"*. https://doi.org/10.5281/zenodo.21624174
[59] Pablo Valle and Aitor Arrieta. 2022. Towards the Isolation of Failure-Inducing Inputs in Cyber-Physical Systems: is Delta Debugging Enough?. In *2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 549–553.
[60] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Applying and Extending the Delta Debugging Algorithm for Elevator Dispatching Algorithms (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1055–1067.
[61] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Automated Misconfiguration Repair of Configurable Cyber-Physical Systems with Search: an Industrial Case Study on Elevator Dispatching Algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 396–408.
[62] Dennis D Wackerly, William Mendenhall, and Richard L Scheaffer. 2008. *Mathematical statistics with applications*. Vol. 7. Thomson Brooks/Cole Belmont, CA.
[63] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 881–892.
[64] Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. An empirical characterization of software bugs in open-source cyber–physical systems. *Journal of Systems and Software* 192 (2022), 111425.
[65] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.
[66] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta debugging microservice systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 802–807.