QoS-aware Metamorphic Testing: An Elevation Case Study

Jon Ayerdi*, Sergio Segura†, Aitor Arrieta*, Goiuria Sagardui* and Maite Arratibel ‡ Mondragon Unibertsitatea*, Universidad de Sevilla †, Orona ‡
*{jayerdi,aarrieta,gsagardui}@mondragon.edu, †sergiosegura@us.es, ‡marratibel@orona-group.com

***Abstract*—Elevators are among the oldest and most widespread transportation systems, yet their complexity increases rapidly to satisfy customization demands and to meet quality of service requirements. Verification and validation tasks in this context are costly, since they rely on the manual intervention of domain experts at some points of the process. This is mainly due to the difficulty to assess whether the elevators behave as expected in the different test scenarios, the so-called *test oracle problem*. *Metamorphic testing* is a thriving testing technique that alleviates the oracle problem by reasoning on the relations among multiple executions of the system under test, the so-called *metamorphic relations*. In this practical experience paper, we report on the application of metamorphic testing to verify an industrial elevator dispatcher. Together with domain experts from the elevation sector, we defined multiple metamorphic relations that consider domain-specific quality of service measures. Evaluation results with seeded faults show that the approach is effective at detecting faults automatically.**

***Keywords*-Cyber-Physical Systems, Elevators, Metamorphic Testing, Quality of Service**

I. Introduction

Elevator installations are Cyber-Physical Systems (CPSs) that must satisfy vertical transportation demands while providing the best possible Quality of Service (QoS) to its users. Nowadays, there is an increasing amount of metrics that can be used to measure the QoS of a system of elevators, such as the Average Waiting Time (AWT) for the passengers or the energy consumption of the elevators [5], [32].

In order to ensure that an elevator installation complies with the customer demands, a thorough verification and validation process must ensure that all the relevant QoS measures are within acceptable boundaries. Unfortunately, this verification and validation process is expensive, as manual intervention of domain experts is required at certain points. Since the dispatching algorithm is thoroughly tested in several different elevators installations, and each of them has different transportation demands and requirements, it is usually difficult to determine the exact QoS measure (e.g., the AWT value) that should be expected from a test, i.e., it is difficult to predict the test outcome. The resulting inability to determine whether a test outcome is correct or not is known as the *oracle problem* [6]. As a consequence, a manual assessment by the test engineer is often needed in order to determine whether the outcome of a test is good or not, which is becoming increasingly costly as these systems become more complex and customizable.

Metamorphic testing [10], [28] alleviates the oracle problem by adopting a singular approach to software testing: instead of verifying the correctness of each individual execution of the program under test, metamorphic testing exploits known input and output relations that should hold among *multiple* executions of the program, the so-called metamorphic relations. Metamorphic testing has been used in many domains, such as machine learning applications, web services, computer graphics, and compilers [11], [26]. This technique has also been successfully applied in the domain of CPSs, such as for testing wireless sensor networks [9], autonomous drones [18], or self-driving cars [33], [37].

Since its introduction back in 1998, most applications of metamorphic testing have focused on the detection of functional faults. More lately, however, some authors have explored applications of metamorphic testing in the context of non-functional testing [8], [15], [29], [30]. Recently, Segura et al. [29], [30] proposed the concept of performance metamorphic testing, where the metamorphic relations are defined in terms of how the performance of the program under test (e.g., execution time) is expected to change when making certain changes in the program's inputs.

In this practical experience report paper, we describe how we applied metamorphic testing for the automated identification of bugs in an industrial elevator dispatcher. Analogously to performance metamorphic testing, we propose metamorphic relations that relate the performance (QoS metrics) of several executions of the dispatcher under test. However, unlike previous work on performance metamorphic testing, and as a novel contribution, our approach uses performance (QoS metrics) as a proxy to detect functional bugs rather than performance bugs. For example, an unexpected value for the AWT may indicate a wrong assignment of the elevators by the dispatcher due to a (functional) bug in the program.

We evaluated the fault-detection capability of the proposed metamorphic relations using mutation testing. The paper describes the steps followed, as well as the lessons learned. Overall, our approach improved the testing process by providing an automated mechanism to identify undesirable behaviours from the dispatcher. However, we found that some manual work and the advice from domain experts were still needed to tune the metamorphic relations in order to achieve the best results. This manual endeavour, however, is an upfront investment that can be compensated by reusing the metamorphic relations during the development of new versions of the product.

The rest of the paper is structured as follows: Section II provides some background on metamorphic testing. Section

III presents our industrial case study and its testing process. Section IV describes the metamorphic relations we propose for the domain of elevation. Section V presents our empirical evaluation. Section VI describes the main lessons learned from our study and its future prospects. Section VII points out the potential threats to the validity of our study. Section VIII presents the related work and points out our contributions to the state of the art. Section IX concludes the paper.

II. Metamorphic Testing

Metamorphic testing (MT) [10], [28] aims to detect bugs by looking at the relations among the inputs and outputs of two or more executions of the program under test, so called *metamorphic relations* (MRs). For example, consider the program $merge(L_1, L_2)$ that merges two lists into a single ordered list without duplicated elements. Checking if the output of the program is correct for two non-trivial input lists would be difficult: this is an instance of the oracle problem. The order of the parameters should not influence the result, which can be expressed as the following MR: $merge(L_1, L_2) = merge(L_2, L_1)$. In this relation, (L_1, L_2) is the *source test case*, and (L_2, L_1)—created by switching the two input lists—is the *follow-up test case*. This metamorphic relation can be instantiated into one or more *metamorphic tests* by using specific input values and checking whether the relation holds, e.g., $merge([a, k, d], [t, m]) = merge([t, m], [a, k, d])$. If the relation is violated, the metamorphic test is said to have failed, indicating that the program under test contains a bug.

MRs can often be defined at a very abstract level, representing not a single relation, but a set of relations. When this happens, relations are referred to as metamorphic relation patterns [25], [27], [36]. Zhou et al. [36] defines a *metamorphic relation pattern* as an abstraction that characterizes a set of (possibly infinitely many) MRs. MR patterns have proved to be very helpful on guiding testers on the identification of MRs. As an example, Zhou et al. [36] proposed a *symmetry* MR pattern, based on the observation that most systems can be observed from different viewpoints from which the system appear the same. For example, an AI-enabled object recognition system should recognize the same objects in a video, regardless of whether it is played forwards or backwards. MR patterns are often defined as incomplete MRs where only the relation among the inputs or the outputs is specified. These are referred to as *metamorphic relation input patterns* [36] and *metamorphic relation output patterns* [27], respectively.

Most of the works on metamorphic testing have focused on the detection of functional faults [11], [26]. Recently, Segura et al. [29], [30] proposed the concept of *performance metamorphic testing*, where MRs are defined in terms of how the performance of the program under test (e.g., execution time) is expected to change when making certain changes in the program's inputs. For example, intuitively, the execution time required to merge two lists should be equal or greater if more elements are added to both lists. This can be expressed as the following (performance) MR: $T(merge(L_1, L_2)) \leq T(merge(L_1 \cup L_3, L_2 \cup L_4))$, where L_3 and L_4 are two lists containing k random items each, with $k > 0$. Research on performance metamorphic testing is thriving with new applications emerging in domains such as code generators [8] and data analytic platforms [15].

III. Elevation Case Study

An elevator is a complex Cyber-Physical System (CPS) where software and hardware interact with the goal of transporting passengers safely and by considering certain QoS measures. Among the components of the elevator installation, the traffic master is in charge of managing the passenger flow. This element is composed of different software modules, including the dispatching algorithm, which decides which elevator should attend each call. The dispatching algorithm has a high impact on the QoS measures of the elevator installation. These QoS measures include, among others, the Average Waiting Time (AWT), which refers to the average time passengers need to wait until they are attended by an elevator, or the Average Time To Destination (ATTD), which refers to the average time passengers wait until they arrive to their destination. More recently, with the goal of providing greener elevator installations, the dispatching algorithms have started to consider energy consumption as a new QoS measure.

The elevators dispatching algorithms are highly complex, as they need to consider several functionalities for a wide range of types of elevators installations. Orona has a large suite of elevators dispatching algorithms, which need constant maintenance in order to address new functional requirements, new QoS measures, legislative changes, bug fixing, hardware obsolescence or system degradation, adaptation to building requirements, etc. When changes are made, Orona has a well established verification and validation process of the dispatching algorithm before deploying the new release in real installations. The overall verification and validation process is shown in Figure 1. In a first stage, tests are executed within a Software-in-the-Loop (SiL) level. The software of the dispatching algorithm is an executable that communicates with a domain-specific simulator named Elevate. Elevate simulates all the physical components of the elevator and provides a set of QoS measure results when the simulation has finished. The following stage is related to the HiL stage. In this case, the software of the dispatching algorithm is integrated with the rest of software and hardware infrastructure, encompassing, among others, real-time operating systems, communication protocols, and the real-target at which the software is executed. At this stage, the tests are executed in real-time, and their goal is to validate the functional correctness of the release within the real infrastructure. Lastly, the software is deployed into the real system at operation. The elevator maintainer performs a set of manual tests to ensure that the software has been successfully deployed and that it works correctly. As the test level becomes more realistic, the test execution cost increases significantly. Therefore, it is important to detect bugs as early as possible during the verification and validation process.

At the SiL and HiL test levels, two types of tests are carried out: (1) short-scenario tests and (2) full-day tests. The former's

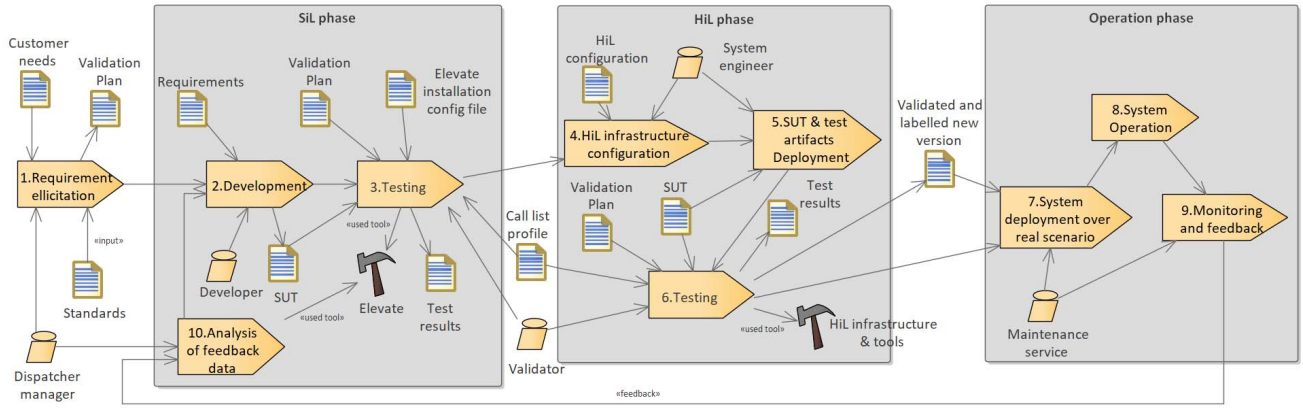

Fig. 1: Process for testing the elevator dispatching system [4]

objective is to test specific functional properties by providing short and isolated scenarios. The simulations performed are usually 1 to 6 minutes long. In the latter, the objective is to test the behavior of a system of elevators by mimicking a normal full-day (or sub-scenarios of it) in the life-cycle of an elevator. These tests simulate the passenger flow traffic of 2 to 18 hours. In this case, to assess whether the test has passed or failed, certain QoS measures (e.g., AWT) are considered (both overall values and values within time-series). This paper is focused on short-scenario tests, where determining the expected test outputs is not evident, and in occasions, infeasible. This is, to a large extent, caused by (1) the dynamic environment in which the elevators operate and (2) the building installation. For the dynamic environment, changing a property of a specific call of a passenger (e.g., its arrival time or its destination floor) can have a drastic impact on the overall behavior of the system of elevators. As for the building installation, the outcome of a test is completely influenced by the elevator's characteristics. For instance, the AWT highly depends on several properties, including the number of elevators, the dynamic properties of each elevator (e.g., speed, acceleration, etc.), the maximum number of passengers each elevator can lift, etc. That is why in most of these short-scenario tests, manual intervention by the test engineer is required to determine whether a test has passed or failed. For this manual intervention, domain experience is also often needed.

A test for the dispatching algorithm is constituted by (1) the test input and (2) the building installation information. The *test input* consists in a list of passengers which arrive to a landing, call an elevator, and request a destination. For each passenger, the following information is provided: (1) the arrival time, (2) the arrival floor, (3) the destination floor, (4) the weight of the passenger, (5) capacity factor by mass, (6) the loading time, (7) the unloading time and (8) how the passenger behaves when not all elevators serve all floors. Regarding the *building installation information*, it refers to an XML file with all related information of the building and elevators installation at which the SUT is being tested. For instance, its information encompasses the number of floors of a building, number of elevators, floors served by each of the elevators, maximum weight each elevator can lift, etc.

Different elevator dispacthers can be used to optimize different objectives depending on the installation requirements and traffic profiles. For example, the dispatcher that we use for our evaluation is based on a rule based algorithm which optimizes for the best AWT.

We assessed several QoS metrics among the ones typically used within the domain in order to evaluate passenger experience, particularly the ones that we expected could help reveal faulty behaviors and inefficient dispatching within test cases. The following are the specific metrics we selected for our MRs after discussing with domain experts:

Average Waiting Time (AWT). This is the average time from the moment a landing call is issued until an elevator stops to attend the call. This metric does not take the transit times for the calls issued from inside the elevator into account. This is among the most important metrics for providing a good user experience [5], and as we previously mentioned, is the metric for which the dispatcher we test is designed to optimize.

Total Distance (TD). This is the sum of the distances traversed by all the elevators of the building. We measure this distance in floors rather than actual distance, although this would not make any difference for our experiments because the building we used has equally spaced landings. We consider this metric as relevant because an unexpected value may reveal behaviours such as consistently not assigning elevators which are close to the landing calls or unnecessarily dispatching multiple elevators to a single call.

Total Movements (TM). This is the sum of all the movements (i.e., engine start-ups) of all the elevators of the building. We considered that this metric may reveal inefficient dispatching or bugged behaviours in a similar way to *Total Distance*. When compared with *Total Distance*, we expected this metric to be easier to predict, although potentially less effective in detecting failures.

IV. Metamorphic Relations

In this section, we describe the MRs proposed for the identification of failures in elevator dispatchers. Specifically,

we propose three metamorphic relation input patterns (MRIPs) and three specific MRs derived from each pattern, each of which is related to one of the QoS metrics described in the previous section. Each MRIP describes an input relation between the source and a follow-up test case exploited in different MRs.

These MRs are defined based on non-functional properties (the QoS metrics) of the system, which in practice makes this approach similar to performance metamorphic testing. However, in contrast to previous work on performance metamorphic testing, our aim is detecting functional failures (i.e., incorrect or inefficient choices from the dispatching algorithm) rather than performance bugs.

The effectiveness of the metamorphic relations can be severely affected by the features of the test cases that are used, such as their duration. For our simulation-based experimental setup, it is not feasible to perform metamorphic testing with day-long traffic profiles, since the time required to execute a significant amount of source and follow-up test cases would be exceedingly long. Therefore, the MRs we present and evaluate in this paper have been designed with short-scenario test cases in mind. We hypothesize that some of the proposed MRs could also be applicable to large test cases, but that should be investigated further.

The execution of each test case is expressed as a call to the operation $serve(E, C)$, where E is the set of initial elevator positions ($|E|$ being the number of elevators), and C is the list of passenger calls, where each $c \in C$ has a source and destination floor, and a timestamp in which the passenger will issue a landing call from the source floor. For the sake of simplicity, the relations used in our work are composed of two test cases, the source test case and one follow-up test case, although they could be easily generalized to two or more follow-up test cases. When needed, we denote the source test case as (E_s, C_s) and the follow-up test case as (E_f, C_f).

The testing based on QoS metrics is inherently complex due to the lack of a clear line between acceptable and failing behaviours. Although the manufacturer usually defines a worst-case value for the QoS metrics, the elevators should perform better than that under most circumstances, so there are many potential failures that cannot be detected by only checking these minimum requirements. This task becomes even harder when we consider that there are many specific scenarios where apparently faulty behaviour is actually correct, such as in cases where there are trade-offs between different QoS metrics. In order to mitigate this issue, inspired by [23], we defined tolerance *thresholds* that allow small differences between the expected and actual results to pass. Therefore, when we express a MR as $serve(E_f, C_f) \lesssim F(serve(E_s, C_s))$, where F can be any formula over the source and follow-up test case inputs and outputs, its actual implementation will have the form of $serve(E_f, C_f) \leq a \cdot F(serve(E_s, C_s)) + b$, where a and b are tolerance values specific to each MR. The specific values of these parameters were defined by consulting domain experts and experimenting with a subset of the test cases in order to ensure that the non-faulty system (the dispatcher we use for our experiments, which is a validated production version) would not violate the MRs. While in most cases the value of a could just be 1, we found that a constant tolerance b was always needed for some corner cases. This was particularly needed for very short test cases, since a lot of these MRs describe expected trends rather than invariants. For simplicity, our descriptions of the MRs will not specify these tolerance thresholds, but will instead use relational operators that express inaccuracy ($\lesssim$ and $\gtrsim$).

Furthermore, in order to differentiate slight deviations and huge differences between the expected value of a QoS metric and the actual value, all of our MRs provide a quantitative verdict which indicates the severity of the failures. These quantitative verdicts are obtained by transforming the MRs with the form $serve(E_s, C_s) \lesssim F(serve(E_f, C_f))$ into $min(F(serve(E_f, C_f)) - serve(E_s, C_s), 0.0)$. An analogous transformation can also be performed for MRs with the form $serve(E_s, C_s) \gtrsim F(serve(E_f, C_f))$. With this transformation, the verdict from a metamorphic test is a decimal value ranging from 0.0 to $-\infty$, with 0.0 indicating a *PASS* verdict and a negative value indicating a *FAIL* that is more severe the closer to negative infinity it is. These quantitative verdicts can be used to direct the attention of the test engineers towards the most severe failures, which generally should be prioritized. Furthermore, the quantitative verdicts can also enable the use of additional techniques that can improve the testing process, such as using falsification-based automatic test case generation [2], [34] in order to find the test cases that are closer to violating MRs, thus increasing the fault detection capability of the test suite and reducing the overall cost of testing. In fact, Segura et al. have already proposed transforming performance metamorphic relations into fitness functions in order to guide search-based testing [29].

Next, we describe the patterns and relations used in our case study:

MRIP$_1$: Additional calls. This pattern represents those relations where the follow-up test case is constructed by adding one or more passenger calls C' to the passenger calls list in the source test case. Formally, the input relation is defined as follows: $C_f = C_s \cup C'$. When this happens, the Total Movements (TM) of the follow-up test case should either be similar or higher than in the source test case. Furthermore, we can set the worst-case cost for executing the additional call ($TM_{worst}(C')$, which is always 2 movements per call, one to reach the call floor and the other to get to the destination) as the upper bound for their increase. For instance, an additional call should cost two additional movements at worst: one for the landing call and one for the car call. This can be represented as the following relation:

$$\begin{array}{l} TM(serve(E, C_f)) - TM(serve(E, C_s)) \gtrsim 0 \\ TM(serve(E, C_f)) - TM(serve(E, C_s)) \lesssim TM_{worst}(C') \end{array} \quad (\texttt{MR1}_{\texttt{TM}})$$

An analogous relation is expected to hold using the Total Distance (TD) metric ($\texttt{MR1}_{\texttt{TD}}$), where $TD_{worst}(C')$ is calcu-

lated as the sum of $max(\text{FLOORS} - c.source, c.source - 1) + |c.source - c.destination|$ for each $c \in C'$ (the first part being the worst-case distance from the attending elevator to the source floor in a building with FLOORS floors and the second part being the distance from the source to the destination floor).

We can also define a similar relation for the Average Waiting Time (AWT), but in this case the worst possible Total Waiting Time (TWT), i.e., the sum of all the waiting times, is calculated and then divided by the total number of calls in order to calculate the maximum change to the AWT, which could be either an increase or a decrease:

$$|AWT(serve(E, C_f)) - AWT(serve(E, C_s))| \lesssim \frac{TWT_{worst}(C')}{|C_f|} \quad (\text{MR1}_{\text{AWT}})$$

where $TWT_{worst}(C')$ is calculated with a linear function which approximates the worst-case time it would take an elevator to arrive for each $c \in C'$, and then summing the times for each call. This is based on the worst-case arrival distance ($max(\text{FLOORS} - c.source, c.source - 1)$, same as in $TD_{worst}(C')$).

MRIP$_2$: Additional elevators. In this pattern, the follow-up test case is constructed by either adding or removing one or more elevators, E', to the set of available elevators, i.e., $E_f = E_s \cup E'$ if we add elevators or $E_f = E_s \backslash E'$ if we remove them. This transformation will affect each of the proposed QoS metrics differently. For simplicity, all of the equations for these MRs have been written under the assumption that the number of elevators is increased, i.e., $|E_f| > |E_s|$. For the opposite cases, the corresponding MRs can be easily inferred by swapping the E_s and E_f occurrences on all the given equations.

For the AWT, we can expect that after adding more elevators the waiting times will remain unchanged in the worst case, or be reduced otherwise, since the algorithm is supposed to optimize for the best AWT and has more resources (available elevators) to work with. Furthermore, we expect that the AWT cannot be reduced by a factor greater than the factor in which the number of elevators has been increased. For instance, if we triple the number of elevators available, the AWT can be reduced up to a third of the original test case:

$$\begin{aligned} &AWT(serve(E_f, C)) - AWT(serve(E_s, C)) \lesssim 0 \\ &\frac{AWT(serve(E_f, C))}{AWT(serve(E_s, C))} \gtrsim \frac{|E_s|}{|E_f|} \end{aligned} \quad (\text{MR2}_{\text{AWT}})$$

As for the TD, increasing the number of elevators can, on the one hand, reduce the distance traversed by the elevators due to the fact that the more elevators there are, the higher the chance of having a closer elevator when a landing call is issued. On the other hand, the distance may also increase because the dispatcher sent multiple elevators for multiple calls in the same direction, increasing the traversed distance in order to reduce the waiting times. For either case, we can define the upper bound for the change of TD based on the factor in which the number of elevators has been increased, similar to the AWT:

$$\frac{|E_s|}{|E_f|} \lesssim \frac{TD(serve(E_f, C))}{TD(serve(E_s, C))} \lesssim \frac{|E_f|}{|E_s|} \quad (\text{MR2}_{\text{TD}})$$

For TM, we expect this metric to remain unchanged or increase when adding more elevators, since unlike TD, the elevators will need to start-up their engines to move regardless of proximity, whereas the potential increase of this metric due to moving multiple elevators in parallel still applies. On the other hand, the number of elevator start-ups (TM) should not increase by a factor higher than the increase in the number of elevators:

$$\begin{aligned} &TM(serve(E_f, C)) - TM(serve(E_s, C)) \gtrsim 0 \\ &\frac{TM(serve(E_f, C))}{TM(serve(E_s, C))} \lesssim \frac{|E_f|}{|E_s|} \end{aligned} \quad (\text{MR2}_{\text{TM}})$$

MRIP$_3$: Initial position change. This pattern represents those MRs where the initial positions of all the elevators E_s are randomly changed to E_f, where $|E_f| = |E_s|$. This transformation could result in either improving, deteriorating or not affecting the QoS metrics, since it is difficult to predict its effects on the test case execution. Nevertheless, we can define an upper bound on the maximum effect that this transformation should have. For TM and TD, we can define this upper bound as the cost for transitioning from the source initial positions E_s to the follow-up initial positions E_f:

$$|TM(serve(E_f, C)) - TM(serve(E_s, C))| \lesssim TM_{worst}(E_s, E_f) \quad (\text{MR3}_{\text{TM}})$$

where $TM_{worst}(E_s, E_f)$ is the worst case scenario cost for moving the elevators from E_s to E_f. Another relation with the same pattern can be instantiated for the TD (MR3_{TD}).

For the AWT, the longer a test case is, the smaller the effect we can expect from this transformation, since it only affects the initial state of the system, while in longer test cases the dominant factors that will affect the AWT will be the passenger calls and the dispatcher algorithm. Because of this, we divide the estimated maximum cost by the count of passenger calls $|C|$:

$$|AWT(serve(E_f, C)) - AWT(serve(E_s, C))| \lesssim \frac{TWT_{worst}(E_s, E_f)}{|C|} \quad (\text{MR3}_{\text{AWT}})$$

V. Empirical Evaluation

In this section, we describe the experimental validation performed to assess the fault-detection capability of the proposed MRs in an industrial elevator dispatcher.

A. Mutant Generation

Mutation testing was employed to assess the fault detection capability of the proposed approach in the context of elevators dispatching algorithms. This approach has been found to be a valid substitute for testing with real faults [16]. Specifically, we created 99 faulty variants (mutants) of the elevator dispatcher with seeded faults. Faults were manually seeded by a domain expert who applied syntactic changes to the source code of the

dispatcher, which is written in the C programming language. Specifically, faults were seeded applying traditional mutation operators, including arithmetic, logical and relational operator mutations [1]. Faults were introduced in a uniform manner throughout the sections of the source code that are relevant in the simulation environment. Notice that although it is not a large number of mutants, we used simulation-based testing to execute the tests, which requires a significant amount of time. In fact, the number of mutants employed in our evaluation is similar or higher to other approaches using simulation-based testing [3], [19], [22].

After executing the test cases generated for the experiments, 10 out of 99 mutants blocked the simulation indefinitely in some test cases. Thus, we opted for discarding these mutants, since such trivial failures could be easily detected by setting a time out. This resulted in a final set of 89 mutants used in our evaluation. Furthermore, all the mutants were reviewed by a domain expert to check that they were not semantically equivalent to the original program.

B. Test Case Generation

As mentioned in Section IV, our empirical evaluation is based on short-scenario test cases, which have a duration of roughly 3 minutes on average. We chose to use this type of test cases as opposed to longer ones because they allow performing a significant amount of metamorphic tests for all the mutants with a reasonable total execution time for our experiments. Source test cases were randomly generated starting from a template project from a real building with 10 floors and 6 elevators. For each generated test case, we selected a random number of elevators (between 2 and 6), a random initial floor for each elevator, and a random passenger list generated by uniformly distributing a number of calls $|C|$ across a fixed time period $[0, T)$. The source and destination floors for each call $c \in C$ were also uniformly selected from the 10 landing positions of the building. We used T values of 10 seconds, 2 minutes, and 4 minutes. As for the number of calls $|C|$, our source test cases have varying densities, ranging from 2 up to 36 calls per minute ($\frac{|C|}{T}$). In total, we generated 140 random source test cases.

Follow-up test cases were generated by applying changes to the input of source test cases, as described in the proposed MRIP, namely:

- *MRIP1. Additional calls.* An additional random call is inserted to the passengers list. The time of the call is random within the range $[0, T_c]$, where T_c is the time of the last call in the source test case. For our experiments, we only add a single additional call, since the number of calls on each test case is small enough for this to have a noticeable effect on their execution.
- *MRIP2. Additional elevators.* The number of available elevators $|E|$ is changed to a different value $|E'|$, which is randomly sampled from the range of possible values $[2, 6]$ excluding $|E|$. If $|E'| > |E|$, the initial positions of the first $|E|$ elevators remain the same, while the additional elevators get randomized. On the other hand, if $|E'| < |E|$, the initial positions remain the same as the first $|E'|$ elevators in E.
- *MRIP3. Initial position change.* The initial positions of all the elevators are shuffled, without changing the number of available elevators.

In total, we generated 1200 different pairs of source and follow-up test cases: 420 for MRIP1, 360 for MRIP2, and 420 for MRIP3. These test cases, plus the initial set of 140 source test cases, were executed against the original dispatcher and the 89 mutants resulting in a total of $(140 + 1200) \times 90 = 120,600$ executions.

It is worth noting that each pair of source and follow-up test cases were used to check several MRs (those derived from the corresponding pattern). Specifically, each pair of test cases resulted in 3 metamorphic tests, since we derive 3 different MRs for each MRIP (one for each QoS metric). Considering this, there are a total of $(3 \times 420) + (3 \times 360) + (3 \times 420) = 3600$ metamorphic tests per mutant, and $89 \times 3600 = 320,400$ metamorphic tests in total for all the mutants.

C. Results and Discussion

The test generation, execution (in simulator) and metamorphic tests have been fully automated with Python scripts, which have been employed for our empirical evaluation.

Our evaluation mainly employed two metrics to determine the effectiveness of our approach: (1) The *mutation score*, which refers to the percentage of mutants killed by the MRs, and (2) the *failure detection ratio*, which is the percentage of metamorphic tests that resulted in a failing verdict. For both of these metrics, a higher percentage is better. Note that we consider a mutant as "detected" or "killed" when one or more of the metamorphic tests resulted in a MR violation.

After executing the test cases, all of the proposed MRs combined killed 74 out of 89 mutants, which results in a **mutation score of 83%**. Nevertheless, the analysis of each individual MR revealed that there is a great disparity between their performances. The original dispatcher was also verified with the proposed MRs and the same test cases, and none of the metamorphic tests yielded any failing verdict for it.

Metamorphic Relation		Mutation Score		
MRIP1	$MR1_{AWT}$	55.1%	79.8%	83.1%
	$MR1_{TD}$	74.2%		
	$MR1_{TM}$	56.2%		
MRIP2	$MR2_{AWT}$	57.3%	59.6%	
	$MR2_{TD}$	14.6%		
	$MR2_{TM}$	5.6%		
MRIP3	$MR3_{AWT}$	14.6%	46.1%	
	$MR3_{TD}$	40.4%		
	$MR3_{TM}$	4.5%		

TABLE I: Mutation scores of MRs.

Table I shows the mutation scores of each individual MR, as well as the aggregate results for each MRIP and the global score. These results show that $MR1_{TD}$ obtained the highest mutation score by a wide margin, while $MR2_{AWT}$, $MR1_{TM}$, $MR1_{AWT}$ and $MR3_{TD}$ also obtained significant scores. This indicates that these MRs are capable of detecting more (types

of) failures than the rest, at least for the seeded faults used in our evaluation.

On the other hand, there were 1147 out of 320,400 metamorphic test failures, which corresponds with a **failure detection ratio of 0.36%**. The fact that only a small percentage of metamorphic tests detected failures suggests that the mutants were not trivial.

Metamorphic Relation		Failure Detection Ratio		
MRIP1	$MR1_{AWT}$	0.64%	0.45%	0.36%
	$MR1_{TD}$	0.45%		
	$MR1_{TM}$	0.26%		
MRIP2	$MR2_{AWT}$	1.64%	0.59%	
	$MR2_{TD}$	0.09%		
	$MR2_{TM}$	0.04%		
MRIP3	$MR3_{AWT}$	0.04%	0.07%	
	$MR3_{TD}$	0.14%		
	$MR3_{TM}$	0.02%		

TABLE II: Failure detection ratio of MRs.

Table II shows the failure detection ratio of the individual MRs, as well as the aggregate results for each MRIP and the total ratio for all MRs. As illustrated, $MR2_{AWT}$ stands out over the rest by having more than twice the failure rate than any other MR. Other than that, all of the MRs from MRP1 have a significantly higher failure rate than the rest of MRs. A higher failure detection ratio means that the same amount of failures could be detected with a smaller amount of test cases, thus reducing the cost of the test executions. However, a high failure detection ratio but a low mutation score means that failures are only detected in a few specific mutants, which might indicate that the MR can only detect some specific types of failures.

Figure 2 shows the metamorphic failure counts for each mutant. Each of the bars represents one of the 89 mutants, and each color on a stacked bar represents a different MRIP. From this diagram, we can see that the MRs derived from MRIP1 reveal metamorphic failures across most of the mutants. Conversely, the relations derived from MRIP2 seem to reveal many more metamorphic test failures on a few mutants, but does not seem to be that effective for the rest of them. This explains why the mutation score of MRIP2 is lower than that of MRIP1, even though its failure detection ratio is higher (particularly for $MR2_{AWT}$). Finally, the results from MRIP3 as a whole seem unremarkable, with no more than 6 metamorphic test failures on any mutant, whereas either MRIP1 or MRIP2 accomplish significantly better results on almost all mutants.

Furthermore, Figure 3 shows the most severe failures for each mutant and MRIP. Section IV describes the meaning of these values, which in a nutshell encode the degree of the MR violation, with a lower value signifying a more severe failure. It is interesting to note that this diagram strongly resembles the one from Figure 2, which indicates that, in general, mutants that display more severe MR violations tend to have a higher number of MR violations as well (and vice versa).

The overall conclusion from these results is that our approach appears to be effective at automatically detecting failures in the dispatcher, thus alleviating the oracle problem. Furthermore, the quantitative verdicts can allow prioritizing the most severe MR violations over less severe failures. The MRIP1 relations seem to be able to detect more (types of) failures, whereas MRIP2 can detect some types of failures more clearly, i.e., with a higher quantity of metamorphic test failures and more severe MR violations. This seems to indicate that the MRIP1 relations have the best potential for detecting more different types of failures. The possible advantages of MRIP2 over MRIP1 are the fact that it might be able to provide useful results without having to set very tight tolerance thresholds for the MRs and without using as many test cases as for MRIP1, which means that these MRs can be used at a lower cost. Finally, even though MRIP3 did reveal some failures, this group of MRs appears to be the least effective among the proposed patterns. As for the QoS metrics, the AWT, as expected, appears to be effective in most cases, although TD seems to obtain better results in some instances, such as having better mutation scores than the other MRIP1 relations (but worse failure detection ratio) and having the best results among the MRIP3 relations. As for TM, it obtained the worst results among the QoS metrics, although the results from $MR1_{TM}$ could be considered good.

VI. Lessons Learned and Future Prospects

Next, we describe the main lessons learned from our study and its future prospects.

Lesson 1 – Cost-effectiveness of the approach: After experimenting with the MRs we proposed, we have concluded that the approach is indeed effective and can be used to detect functional failures in the elevator dispatchers. Even though the cost of this approach appears to be high, the fact that it can be fully automated makes it a valuable solution. It must be noted that the high cost of this approach comes mainly from the system executions on the simulator, and the time required to check the MRs based on the execution results was negligible in comparison. It may seem that metamorphic testing is more costly than other techniques due to the need for multiple test cases in order to check a MR. However, we must note that a single source test case can be reused for several metamorphic tests, and in our case we used 1200 follow-up test cases based on solely 140 source test cases. Furthermore, having multiple MRs for each MRIP is an advantage because the same test case results can be reused for all of them. This implies that defining and using MRs for additional QoS metrics would be nearly zero-cost as long as they are based on the same MRIPs.

In this regard, we must note that some MRIPs are more flexible than others in terms of the number and the diversity of the follow-ups that can be generated for a source test case. For instance, MRIP1 (Additional calls) enables the generation of as many follow-up test cases as desired, if we consider all the combinations of possible source and destination floors for the call and the time when the call is issued. On the other hand, MRIP2 (Additional elevators) is more restrictive, since there are usually limited possibilities in the number of available elevators. In our case, the number of active elevators could only be a number in the range $[2, 6]$, which limited the

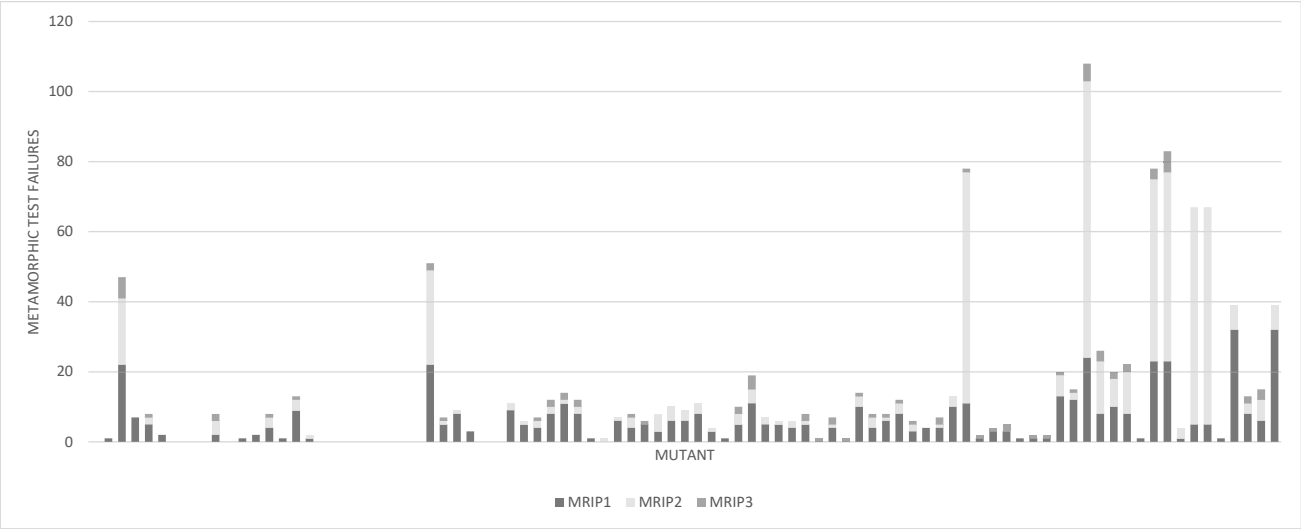

Fig. 2: Metamorphic test failure counts for each mutant.

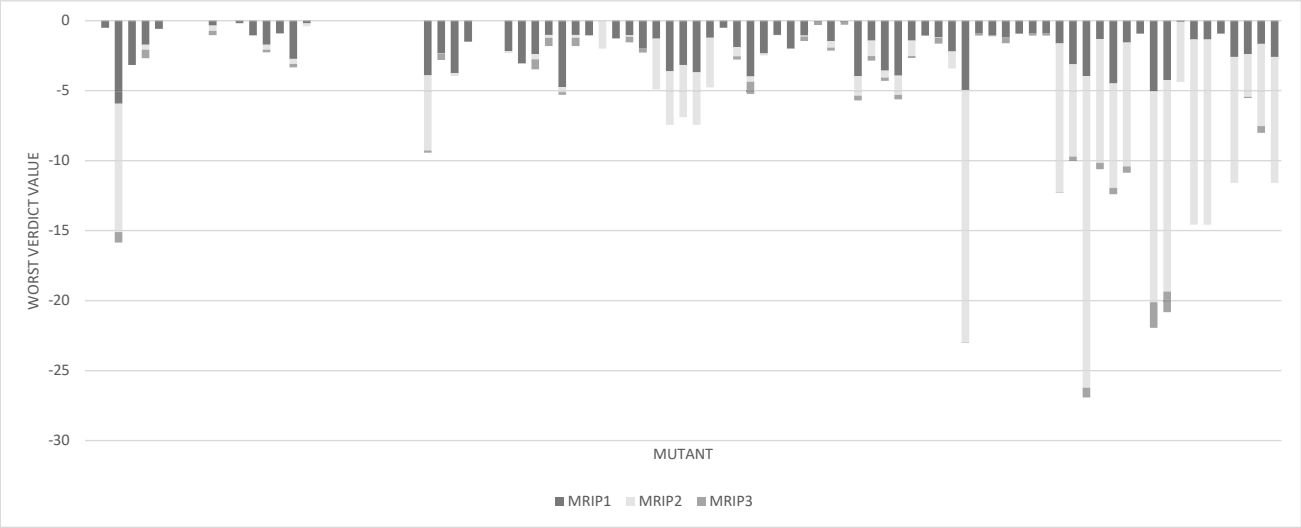

Fig. 3: Worst verdicts for each mutant.

number of follow-ups that could be generated for each source test case to only 4. Finally, MRIP3 (Initial position change) also has a limited number of possible follow-ups, but this is the number of possible elevator positions, which will usually be larger than what will be needed in practice.

Lesson 2 – Disparity between MRs: When we evaluated the effectiveness of our approach, we discovered that there is a great disparity between the performances of the different MRs we proposed. We also found out that one of the MRIPs is somewhat effective with any of the QoS metrics we use, whereas the other two MRIPs only appear to be effective when combined with specific QoS metrics. We can enumerate some of the reasons why some of the MRs may be ineffective: In some cases the output relations might be too loose, and therefore only very severe failures can be detected; in other cases, the effect of the input transformation over the QoS metric might not be relevant for any of the common failure modes of the system. Therefore, new MRs should be tested (e.g., via experimentation with seeded faults) before deciding whether to keep or discard them.

Lesson 3 – Implementation of MRs: The initial definition of the MRIPs was fairly simple and remained mostly unchanged in our implementation. The definition of the specific MRs was more challenging, because understanding the effect that the MRIPs were expected to have on each QoS metric (i.e., the metamorphic output relations) required some knowledge about the domain and the particular system we were testing. Finally, the MRs also required some tolerance thresholds in order to handle some cases where failing verdicts would be undesirable. This is because many of the MRs are not absolute invariants, and although they should hold in general, they may be violated by a small margin on very short test cases or other

uncommon scenarios. Defining these threshold values not only required the assistance from domain experts, but also some manual experimentation in order to discover the worst possible cases that should be tolerated. Previous works on metamorphic testing have also concluded that defining good metamorphic relations requires practical experience in the domain [26]. This manual experimentation, however, is already common in practice, and does not require a significant extra effort from practitioners.

Future prospect 1 – Generalizability to other CPSs: It would be interesting to analyze if the MRIPs and the QoS metrics we have employed in this work can be adopted for other systems. On the one hand, there are elevator dispatchers which have significantly different features than the one we have used for our evaluation, such as those which can optimize for multiple objectives or use non-deterministic algorithms, and the MRs we proposed would need to be adapted for them. On the other hand, the MRIPs we have proposed could be generalized and adapted to other domains. For instance, "Additional Calls" and "Additional Elevators" can be generalized as "Additional Jobs" and "Additional Workers", and then adapted to other systems where a set of workers are assigned jobs that need to be executed efficiently, e.g., a task scheduler which assigns running processes to the available CPUs.

Future prospect 2 – Variability in MRs: Elevators are highly configurable systems, and their software has hundreds of parameters that need to be tuned for each specific installation. Currently, our MRs have been defined in a generic way that could be applicable under most configurations of the tested dispatcher. However, the effectiveness of the test oracles could be further optimized by tuning them for the specific configuration that is being used, such as adjusting the tolerance thresholds and enabling or disabling specific MRs depending of the dispatcher configuration. In this regard, it could be interesting to leverage software product line [24] techniques such as feature models [7] in our MRs.

Besides tuning the test oracles, another possibility could be defining MRIPs based on changes in the dispatcher configurations. For instance, multi-objective dispatchers have parameters that allow tuning the weight of each QoS metric to optimize, indicating its degree of importance, and by changing these weights for the follow-up test case should cause the QoS metrics with increased weights to improve, while the ones with reduced weights might become worse.

Future prospect 3 – Test case generation: The number, complexity and duration of the test case executions is one of the most important factors that determines the cost of simulation-based testing. In order to minimize the time spent on simulation without reducing the effectiveness of the testing process, it is important to have a test suite with a high failure detection ratio. For our approach thus far we have only employed random test case generation, which might not result in a very effective test suite. In order increase the failure detection ratio and reduce the cost of testing, we could exploit the quantitative results provided by our MRs in order to search for test cases that violate them as severely as possible. Even though this process also requires a large upfront investment in simulation time, the resulting test suite would be efficient and reusable, thus reducing the testing cost in the longer run. Similar approaches have already been used in order to search for test inputs that violate some given properties [2], [34].

Future prospect 4 – Fault localization: The experiments performed so far have only considered fault detection, but the subsequent fault localization process has not been addressed yet. It would be interesting to investigate if there is a relation between the quantitative verdict for each MR and the location of the fault in the source code, which could be leveraged in order to automate this process at least partially.

VII. Threats to Validity

A. Internal validity

One of the potential threats to the validity of this empirical evaluation is related with the manual generation of mutants, which might have introduced a bias in our results. In order to mitigate this, we introduced mutations uniformly throughout the relevant parts of the source code, and we generated an amount of mutants which is comparable to other research works that use simulation-based mutation testing [3], [19], [22]. Furthermore, these mutants were also checked in order to identify and filter out equivalent mutants.

Another potential threat is that the randomly generated test cases might have been too few to demonstrate the effectiveness of our approach. We mitigated this by generating a large amount of test cases, which resulted in over a month of execution time for all the generated mutants. This corresponds with several hours of simulation time for a single system, which is comparable to other testing approaches already in use within the domain. We also diversified our test cases in order to include different test case lengths and traffic densities. It should also be noted that these results are all based on short-scenario level tests, with durations of up to several minutes. The effectiveness of the proposed techniques might be completely different if longer test cases (e.g., full day traffic profiles) were to be used instead, which is also a common practice for testing systems in the elevation domain.

Finally, we must also consider the threat introduced by the tolerance thresholds that have been introduced to the MRs. We determined these threshold values by experimenting and consulting with domain experts, and selected values which are high enough to never cause a MR violation on Orona's dispatcher. However, as shown in Figure 3, many mutants have very severe failing verdicts, which indicates that this technique can still detect some faults by just setting naively high tolerance values. For instance, if we were to double the tolerance values for all of the MRs we used in our experiments, the total mutation score would be 50.6% (45 out of 89 mutants killed). Nevertheless, we acknowledge that obtaining the best results from this approach requires some manual experimentation and domain knowledge. The benefit provided by this technique is that the resulting test oracle can be reused for automated testing throughout the rest of the dispatcher's life-cycle.

B. External validity

The main external validity threat relates to the used case study. Although only a single case study was used, it is important to note that it is a real industrial case study, which provides a high degree of complexity to our evaluation. Furthermore, the used dispatching algorithm is the most commonly used one in Orona's elevators. Nevertheless, we acknowledge that our results may not be applicable to other systems in the domain with different features from the one employed in our evaluation, such as elevator dispatchers that are capable of optimizing for multiple objectives, or dispatchers that employ evolutionary algorithms rather than deterministic algorithms.

VIII. Related Work

Metamorphic testing has been used as a solution to mitigate the oracle problem in many types of cyber-physical systems. In [18], metamorphic testing and model based testing approaches are combined in order to test autonomous drones in a simulated environment. Several other recent publications have also proposed the use of metamorphic testing in order to verify autonomous self-driving cars [33], [37]. This is a particularly difficult task because these systems are typically based on machine learning models, and therefore predicting their expected output is often infeasible. However, to the best of our knowledge, this work constitutes the first research publication concerning the application of metamorphic testing in the domain of elevation.

To this date, the majority of metamorphic testing-related works use MRs that are related with functional properties, whereas non-functional properties have only been applied sporadically. As an early example of using non-functional properties, Chan et al. addressed testing wireless sensor networks with metamorphic testing, and they proposed a MR based on the power consumption of the computations from the wireless sensors [9]. More recently, Segura et al. discussed performance metamorphic testing [29], [30] as a mostly unexplored research topic and identified its potential advantages and challenges. Besides performance-related properties, metamorphic security testing is also a type of non-functional testing that is being explored [12], [20]. Lately, performance metamorphic testing has been adopted in new domains, such as webpages [15] and code generators [8]. Nevertheless, the application of performance metamorphic testing has not been explored yet in many domains, and most of the existing examples of this approach address testing software applications rather than cyber-physical systems. Besides the application of metamorphic testing in a new domain, our main contribution to the state of the art is the usage of non-functional properties (QoS metrics) in order to identify functional failures, which has not yet been explored extensively by any previous work. In fact, to the best of our knowledge, [9] contains the only instance of such MRs being suggested on a research paper.

When it comes to performance failure detection, most approaches so far rely on either setting threshold values which can never be violated, detecting known types of problems, or comparing the results against existing data [14]. In some cases, it is possible to execute the same performance tests multiple times in order to detect inconsistencies among executions, or even performing comparisons within the results of the same execution [14], but the most common approach is to perform regression testing against an existing baseline (e.g., a previous version of the system) [13], [14], [31]. Nevertheless, these solutions require an appropriate baseline in order to evaluate the performance results, which may not be available in some cases. A solution that can can mitigate the shortcomings of regression testing are machine learning based techniques. For instance, [21] employs several supervised and unsupervised machine learning approaches for generating performance signatures and detecting deviations. However, these techniques still require appropriate training data, which is not an issue with metamorphic testing.

As for examples of the industrial adoption of metamorphic testing, this technique has been successfully applied to the Data Collection JavaScript Library of the Adobe Analytics software in order to find bugs related with specific versions of browsers or their JavaScript engines [35]. In [17], a model-based metamorphic testing approach is used in NASA's Data Access Toolkit, which is an interface to query a large database of telemetry data, in order to verify that its API returns the correct data for the input queries. This work introduces the use of metamorphic testing in the industrial domain of elevation.

IX. Conclusions

Metamorphic testing based on QoS metrics is a promising solution for alleviating the oracle problem in the domain of elevation. In this paper, we have proposed several Metamorphic Relation Input Patterns (MRIPs) and QoS metrics to use for testing elevator dispatchers, and we have derived specific MRs for the most commonly used elevator dispatcher from Orona. By employing mutation testing on this case study, our experiments have concluded that many of these MRs can detect a high percentage of the randomly injected faults, with 83% (74 out of 89) of the mutants being killed in total. Even though the cost of this approach appears to be high, the reusing of test cases, and possibly the adoption of better test generation techniques, can mitigate this issue by reducing the time spent on simulations.

Acknowledgment

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319. Jon Ayerdi, Aitor Arrieta and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1326-19), supported by the Department of Education, Universities and Research of the Basque Country.

This work has also been partially supported by the European Commission (FEDER) and Spanish Government under projects APOLO (US-1264651), HORATIO (RTI2018-101204-B-C21), and EKIPMENT-PLUS (P18-FR-2895).

REFERENCES

[1] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue . . . , 1989.

[2] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

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

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

[5] Gina Barney and Lutfi Al-Sharif. *Elevator traffic handbook: theory and practice*. Routledge, 2015.

[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[7] David Benavides, Alexander Felfernig, José A Galindo, and Florian Reinfrank. Automated analysis in feature modelling and product configuration. In *International Conference on Software Reuse*, pages 160–175. Springer, 2013.

[8] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry. Leveraging metamorphic testing to automatically detect inconsistencies in code generator families. *Software Testing, Verification and Reliability*, 30(1):e1721, 2020. e1721 stvr.1721.

[9] WK Chan, Tsong Y Chen, Shing Chi Cheung, TH Tse, and Zhenyu Zhang. Towards the testing of power-aware software applications for wireless sensor networks. In *International Conference on Reliable Software Technologies*, pages 84–99. Springer, 2007.

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

[11] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4:1–4:27, January 2018.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55, 2016.

[13] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 159–168. IEEE, 2015.

[14] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[15] O. Johnston, D. Jarman, J. Berry, Z. Q. Zhou, and T. Y. Chen. Metamorphic relations for detection of performance anomalies. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 63–69, 2019.

[16] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[17] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Árdal, and Robert E Wiegand. Metamorphic model-based testing applied on nasa dat–an experience report. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 129–138. IEEE, 2015.

[18] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 35–41. IEEE, 2017.

[19] Bing Liu, Shiva Nejati, Lionel C Briand, et al. Improving fault localization for simulink models using search-based testing and prediction models. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 359–370. IEEE, 2017.

[20] Phu X Mai, Fabrizio Pastore, Arda Goknil, and Lionel Briand. Metamorphic security testing for web systems. *13th IEEE Conference on Software Testing, Validation and Verification, ICST 2020*, 2020.

[21] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th international conference on software engineering (ICSE)*, pages 1012–1021. IEEE, 2013.

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

[23] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 189–200, New York, NY, USA, 2009. ACM.

[24] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[25] S. Segura. Metamorphic testing: Challenges ahead (keynote speech). In *Proceedings of the 3rd International Workshop on Metamorphic Testing (ICSE MET'18)*, New York, NY, USA, 2018. ACM. Slides available at http://personal.us.es/sergiosegura/files/presentations/segura18-MET.pdf.

[26] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, Sept 2016.

[27] S. Segura, J.A. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic testing of RESTful Web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, Nov 2018.

[28] S. Segura, D. Towey, Z.Q. Zhou, and T.Y. Chen. Metamorphic testing: Testing the untestable. *IEEE Software*, 37(3):46–53, 2020.

[29] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés. Performance metamorphic testing: Motivation and challenges. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pages 7–10, 2017.

[30] Sergio Segura, Javier Troya, Amador Durán, and Antonio Ruiz-Cortés. Performance metamorphic testing: A proof of concept. *Information and Software Technology*, 98:1 – 4, 2018.

[31] Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 15–26, 2015.

[32] Marja-Liisa Siikonen. On traffic planning methodology. *Elevator technology*, 10:267–274, 2000.

[33] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314. ACM, 2018.

[34] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1555–1562. IEEE, 2018.

[35] Zhenyu Wang, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. Metamorphic testing for adobe analytics data collection javascript library. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, pages 34–37, 2018.

[36] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey. Metamorphic relations for enhancing system understanding and use. *IEEE Transactions on Software Engineering*, 2018.

[37] Zhi Q Zhou and Liqun Sun. Metamorphic testing of driverless cars. 2019.