Generating Metamorphic Relations for Cyber-Physical Systems with Genetic Programming: An Industrial Case Study

Jon Ayerdi
jayerdi@mondragon.edu
Mondragon University
Mondragon, Spain

Valerio Terragni
v.terragni@auckland.ac.nz
University of Auckland
Auckland, New Zealand

Aitor Arrieta
aarrieta@mondragon.edu
Mondragon University
Mondragon, Spain

Paolo Tonella
paolo.tonella@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Goiuria Sagardui
gsagardui@mondragon.edu
Mondragon University
Mondragon, Spain

Maite Arratibel
marratibel@orona-group.com
Orona
Hernani, Spain

ABSTRACT

One of the major challenges in the verification of complex industrial Cyber-Physical Systems is the difficulty of determining whether a particular system output or behaviour is correct or not, the so-called test oracle problem. Metamorphic testing alleviates the oracle problem by reasoning on the relations that are expected to hold among multiple executions of the system under test, which are known as Metamorphic Relations (MRs). However, the development of effective MRs is often challenging and requires the involvement of domain experts. In this paper, we present a case study aiming at automating this process. To this end, we implemented GAssertMRs, a tool to automatically generate MRs with genetic programming. We assess the cost-effectiveness of this tool in the context of an industrial case study from the elevation domain. Our experimental results show that in most cases GAssertMRs outperforms the other baselines, including manually generated MRs developed with the help of domain experts. We then describe the lessons learned from our experiments and we outline the future work for the adoption of this technique by industrial practitioners.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Theory of computation** → **Assertions**; • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Genetic programming**.

KEYWORDS

cyber physical systems, metamorphic testing, quality of service, oracle generation, oracle improvement, evolutionary algorithm, genetic programming, mutation testing

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3473920

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

1 INTRODUCTION

Cyber-Physical Systems (CPSs) are complex systems that integrate both physical and software components [3, 8, 25]. While the controller of a CPS may run discrete software, the physical layer is composed of parallel physical processes running in continuous time. These types of system are found in many domains, such as aerospace, automotive, healthcare and consumer appliances [23].

The problem of determining whether a test outcome is correct or not is known as the *oracle problem* [10]. The complexity of CPSs and their requirements, combined with the uncertainty of their interactions with the physical world (environmental conditions, user behaviour, etc.), makes the definition of effective test oracles especially challenging [22].

An example of complex CPSs is the traffic manager of a system of elevators developed by Orona [30], one of the leading elevators companies in Europe. Elevator installations are CPSs that must satisfy vertical transportation demands while complying with specific customer requirements and providing the best possible user experience. The compliance with these requirements can be determined by several Quality of Service (QoS) metrics that are measured during the system's operation. For instance, the Average Waiting Time (AWT) for the passengers is known to be one of the most important factors related with user satisfaction [9]. This type of systems needs to pass a thorough verification and validation process in order to ensure its compliance with both functional and non-functional requirements, which involves the intervention of domain experts at many points of the process. Unfortunately, it is often difficult to determine the exact QoS measure that should be expected from a test because the test executions are highly dynamic and sensitive to the timing of physical events (e.g., pressing the elevator call buttons) and communications between components.

Metamorphic testing [14, 35] alleviates the oracle problem by checking whether multiple test executions fulfill certain necessary properties called Metamorphic Relations (MRs) [14]. More specifically, instead of verifying the correctness of each individual execution of the program under test, metamorphic testing exploits known input and output relations (called MRs) that should hold among *multiple* executions of the program. Metamorphic testing has been

used in many domains, such as machine learning, web services, computer graphics, and compilers [16, 33]. Recently, Facebook has adopted it for their simulation and testing infrastructure in order to tackle the test oracle problem, as well as test flakiness [2].

Metamorphic Testing has also been successfully applied in the domain of CPSs, e.g., for testing wireless sensor networks [13], autonomous drones [26], or self-driving cars [42, 47]. Some researchers have also investigated metamorphic testing in the context of non-functional properties, such as performance or QoS metrics [5, 11, 21, 36, 37].

Recently, Ayerdi et al. proposed an approach based on metamorphic testing that considered QoS metrics from the elevation domain at Orona, showing promising results [5]. However, the definition of effective MRs was possible only with in-depth knowledge of the domain and the system under test [15, 33]. As a result, the development and maintenance of effective MRs required a heavy involvement of domain experts who have extensive experience with the system, which is a high cost to pay.

A possible solution to reduce the cost of designing MRs is their automatic generation based on samples of *correct* and *incorrect* execution of the system under test. Such samples could be obtained from a curated database of real test executions, possibly complemented by mutation testing. Our goal is to automatically generate metamorphic relations by minimizing the number of false positives (FPs) over the *correct* samples and the number of false negatives (FNs) over the *incorrect* ones, so as to obtain a metamorphic oracle that can predict the correctness of an outcome as accurately as possible. This solution was inspired by recent advances in the automated improvement of program assertions [18, 40].

In this paper, we present an industrial experience report on automatically generating MRs. Towards this goal, we implemented **GAssertMRs**, **Genetic ASSERTion improvement for MRs**, a tool to generate MRs automatically with genetic programming. GAssertMRs is an adaptation to metamorphic testing and CPSs of GAssert [39–41], a technique for generating and improving program assertions. GAssertMRs formulates the oracle generation and improvement process as a multi-objective optimization problem [38] with three objectives: (i) minimizing the number of FPs, (ii) minimizing the number of FNs, and (iii) minimizing the size of the generated MRs. GAssertMRs prioritizes assertions with fewer false positives, as manually removing them is an expensive activity.

In this experience paper, we evaluate GAssertMRs in the context of an industrial case study from the elevator domain provided by Orona. The study aims to assess the effectiveness of our approach for generating MRs based on the QoS measures obtained during the test executions. Our results show that the automatically generated MRs are effective at detecting faults. Moreover, they are comparable or even outperform manually-generated MRs in almost all cases. In summary this paper makes the following contributions:

- reports an industrial experience of applying a prototype tool to automatically generate MRs in the context of CPSs;
- presents GAssertMRs, an adaptation of GAssert to automatically improve/generate MRs;
- discusses the challenges and lessons learned from the perspective of an industrial setting.
- releases the executable of the tool and the results [6]

2 INDUSTRIAL CASE STUDY

Our case study is a traffic manager of a system of elevators developed by Orona [30]. The traffic manager is composed of different software modules. One of the most important ones is the *Dispatching Algorithm*. This algorithm decides which elevator should serve an elevator call by considering different aspects, including the estimated passengers' Average Waiting Time (AWT).

Orona constantly adds new functionalities to the dispatching algorithm to offer a solution that satisfies all customers' demands, such as reducing energy consumption or giving higher priority to certain elevator calls during emergency situations (e.g., for critical situations in hospitals). The dispatching algorithm is a critical component in a system of elevators. Indeed, a wrong assignment of calls can have a huge impact on the overall Quality of Service (QoS) [5]. Orona has developed a large corpus of dispatching algorithms, making it necessary to employ cost-effective and automated testing solutions for their verification and validation.

The dispatching algorithms of Orona are implemented in C/C++ and their tests are executed using simulation-based testing. Specifically, in this paper we have considered tests at the Software-in-the-Loop (SiL) test level, executed through a domain-specific tool named Elevate.

A test for the dispatching algorithm in Elevate consists of (i) the passenger list, and (ii) the building installation information. The *passenger list* represents a list of passengers that arrive to a landing floor, call an elevator, and request a destination. For each passenger, information such as the arrival time, arrival floor, destination floor, and weight of the passenger is provided in a file.

The *building installation information* is an XML file containing data related to the building and to the elevators installation. Among others, such information encompasses the number of floors of a building and its population, number of elevators and their initial positions, floors served by each of the elevators, and maximum weight each elevator can lift.

For each test execution, Elevate returns the values of domain specific QoS metrics (e.g., AWT, energy consumption). In our study, we considered the three QoS metrics used in our previous work [5]:

1) Average Waiting Time (AWT), which refers to the average time the passengers waited from the time they arrive to the calling floor until the elevator arrived and opened the door. According to the study of Barney and Lutfi [9], AWT is the most important measure to determine whether a system of elevators performs well or not from the perspective of the elevator passengers.

2) Total Distance (TD), which refers to the sum of the distances traveled by all the elevators of the building, measured in floors [5]. This metric is relevant because an unexpected value may reveal faulty behaviours like consistently not assigning elevators close to the landing calls or unnecessarily dispatching multiple elevators to a single call.

3) Total Movements (TM), which refers to the total number of engine start-ups of all the elevators of the building [5]. Similar to the previous metric, this QoS measure may reveal inefficient dispatching or faulty behaviours.

Elevators dispatching algorithms are generally very complex, as they need to consider several functionalities for a wide range and types of elevators installations. In addition, these algorithms are

highly configurable to accommodate all kinds of possibly unique building installations. For example, each building has a unique combination of number of elevators, number of floors, distance between floors, elevator speed, and passenger capacity. Because of this, and due to the dynamic nature of the environment in which elevators operate, it is extremely difficult to determine the correct output of a test. Hence, we resort to the QoS metrics mentioned above to expose misbehaviours or suboptimal behaviours.

3 METAMORPHIC TESTING

Software testing aims at exposing software failures by executing test cases. A test case consists of a test input, to exercise the program under test, and a test oracle, to decide whether the test case execution exposes a software failure. Often, test oracles compare the actual output of the test case with the expected one [10]. To define such test oracles, developers need to specify the expected output for each test case, which can be difficult and error-prone, especially for programs with complex outputs. This problem is called the oracle problem and it is recognised as one of the fundamental challenges in software testing [10].

Metamorphic testing [14] is a technique aiming to alleviate the oracle problem. It is based on the intuition that often it is simpler to reason about relations between the inputs/outputs of multiple, related test executions, rather than the relations between inputs/outputs of each individual test execution.

Given two test cases defined by their respective inputs and outputs, $TC_1 = \langle I_1, O_1 \rangle$, $TC_2 = \langle I_2, O_2 \rangle$, whenever a given input relation r_I holds between the two inputs, a corresponding output relation r_O is expected to hold between the outputs. Hence, the metamorphic oracle can be expressed as:

$$r_I(I_1, I_2) \Rightarrow r_O(O_1, O_2)$$

Any transformation $I_1 \rightsquigarrow I_2$ that satisfies $r_I(I_1, I_2)$ is called a Metamorphic Relation Input Pattern (MRIP) [48].

Let us consider an example from the elevation domain. Given a source test case $TC_1 = \langle I_1, O_1 \rangle$, we can introduce a small alteration, so as to obtain a new test scenario (follow-up test case $TC_2 = \langle I_2, O_2 \rangle$) that differs from the previous one only by the addition of one extra passenger performing an elevator call (hence, r_I prescribes that I_1 and I_2 differ by just one passenger call).

Then, to define the output relation r_O we use the Total Distance (TD) metric, defined as the sum of the distances traversed by all the elevators in the elevation system. The following output relation r_O is expected to hold when the input relation r_I holds:

$$TD_f \geq TD_s + D_{wait} + D_{travel} - 20.14$$

where TD_s and TD_f are the TD values for the source and follow-up test cases respectively, D_{wait} is the longest possible distance to the calling floor of the additional call, and D_{travel} is the distance from the calling floor to the destination floor of the additional call. This particular output relation, including the constant 20.14, was automatically generated by GAssertMRs and is part of the evaluation results.

This automatically generated Metamorphic Relation (MR) specifies that the TD of the follow-up test case (TD_f) should be no less than the TD of the source test case (TD_s) plus the worst-case distance to reach the calling floor and the distance between the calling and the destination floor minus a constant factor that accounts for the optimizations performed by the elevator scheduler. A violation of this MR could indicate that TD_s is abnormally large wrt TD_f, hinting at a possibly faulty behaviour by the system during the source test case execution.

4 APPROACH

Recently, Terragni et al. proposed GAssert [40], a technique to automatically improve assertion oracles. Assertion oracles (also known as program assertions) are executable Boolean expressions that predicate on the values of variables at specific program points.

An assertion oracle should pass (return true) for all correct executions and fail (return false) for all incorrect executions. However, because designing assertion oracles is difficult, they often suffer from both false positives and false negatives [18].

Definition 1. *A **false positive (FP)** of an assertion is a correct program state in which the assertion fails (but should pass).*

Definition 2. *A **false negative (FN)** of an assertion is an incorrect program state in which the assertion passes (but should fail).*

False positives and false negatives are problematic in assertion oracles, as they trigger false alarms and ignore failures [18, 19].

The oracle improvement process of GAssert takes as an input an assertion to improve, and an initial set of correct and incorrect program states. It obtains the correct states by executing a test suite on an instrumented version of the program under test that collects the values of variables at the specified program point. It obtains the incorrect states by executing the same test suite on a series of mutated versions of the program under test with seeded faults (i.e., mutations) [20]. GAssert then explores the space of possible assertions with a co-evolutionary algorithm guided by fitness functions that reward solutions with fewer FPs and FNs.

Evolutionary algorithms [7] are population-based meta-heuristic optimization algorithms, inspired by the mechanisms of biological evolution: selection, reproduction, and mutation. Such algorithms evolve a population of candidate solutions to the optimization problem. In GAssert, candidate solutions are Boolean expressions composed of numerical and Boolean variables.

The algorithm returns an *improved* assertion with zero FPs and the lowest number of FNs (with respect to the correct and incorrect states in input). GAssert favours assertions with zero false positives, as false alarms are usually quite expensive to debug.

Given an improved assertion, the tool OASIs [18] obtains new FPs and FNs for the improved assertion by combining test generation and mutation testing. The resulting program states are added to the sets of correct and incorrect states for the next iteration of GAssert.

The oracle improvement process of GAssert terminates when OASIs does not find new false positives and false negatives for the improved assertion or the global time budget expires.

4.1 Extension to Metamorphic Testing

GAssert was demonstrated to be effective in improving assertion oracles [40]. This motivated us to extend the GAssert approach to generate effective metamorphic relations that minimize the number of false positives and false negatives. Indeed, metamorphic relations

Table 1: Features of Elevation Test Cases.

Feature Name	Description
ElevatorsCount	Count of elevators available
ElevatorsDistanceFloors	Sum of the distances between the positions of each elevator in the source and follow-up test cases (MRIP3 only)
ElevatorsDifferenceTime	Aproximated time it takes to traverse ElevatorsDistanceFloors (MRIP3 only)
PassengersCount	Count of passenger calls
PassengersWaitFloors	Maximum number of floors that must be traversed to reach the calling floor of each call
PassengersWaitTime	Approximated time it takes to traverse PassengersWaitFloors
PassengersTravelFloors	Distance in floors from the calling floor to the destination of each call
PassengersTravelTime	Approximated time it takes to traverse PassengersTravelFloors

are a specific kind of Boolean expressions that predicate on the input/output relations of test cases, instead of predicating on internal variables like assertion oracles. We call this extension GAssertMRs. This extension employs the same oracle generation/improvement algorithm as GAssert, adapting it to the context of metamorphic testing. We now describe in detail the differences between GAssert and GAssertMRs, and how GAssertMRs generates MRs.

The first major difference is that GAssertMRs is not implemented as a part of a MRs iterative improvement process. In other words, there is no outer loop for oracle assessment, which generates new test cases and mutations. While GAssert relies on OASIs [18] for the assessment of assertion oracles, no equivalent tool has been developed yet for MRs. This means that the MRIPs must have been defined before running GAssertMRs, since GAssertMRs only automates the generation/improvement of the output relations.

The second major difference is that in this work we focus on black-box testing of systems, in which oracles predicate on the inputs and the outputs of the entire system. In contrast, GAssert was originally developed for program assertions, which predicate on the internal variables of a single unit under test [40].

Furthermore, one of the key differences between MRs and regular assertions is that MRs are defined over the inputs and outputs of two or more test cases. In fact, a MR oracle consists of a relation between the outputs produced by the source and the follow-up test cases. Because of this, the generated expressions should predicate on the inputs and outputs of the source and follow-up test cases, as opposed to predicating over the variables of an individual test case.

GAssert generates arbitrary Boolean expressions, which would be inadequate in the context of MRs. This is because an arbitrary Boolean expression might not define a relation between the outputs of the source and the follow-up test cases, i.e., the result might not be a MR. For this reason, we restrict GAssertMRs to generate Boolean expressions with the following form:

$$O_f \ [operator] \ F(O_s, I_s, I_f) \qquad \text{(MR Template)}$$

where O_s and O_f are the outputs obtained in the source and follow-up test case executions; I_s and I_f are the inputs of the test cases; $F(O_s, I_s, I_f)$ is a numerical expression; and, $[operator]$ is a relational operator (such as $=, \neq, <, >, \leq$ or $\geq$).

In our implementation of the tool, we assume that the inputs and outputs all have numeric types. For cases where not all inputs are numeric, which is also the case of our elevation case study, a domain-specific function which extracts numeric features from the inputs/outputs should be defined. Table 1 shows the features we use for the test cases in the elevation domain, where the inputs are a set of elevators and their positions and a passenger calls list. Which features to include in the generated MR depends on the inputs affected by the MRIP being used: if an MRIP does not change a particular input, all the variables related with that input are redundant and can be safely ignored. For instance, there is no need to include the count of passenger calls (feature *PassengersCount*) for both the source and the follow-up test cases if the MRIP does not change the passenger calls list.

The generated MRs use only a single output variable, which must be numeric. Thus, to consider different output variables, a separate execution of GAssertMRs is needed for each of them, and a different MR will be generated for each output variable.

To enforce the template described above, when GAssertMRs explores the space of possible MRs it only generates numerical expressions in the form $F(O_s, I_s, I_f)$, whereas the selected output variable (O_s) and relational operator ($[operator]$) will be input parameters for the tool. In this way, each individual of the populations of the evolutionary algorithm is a numeric expression $F(O_s, I_s, I_f)$. Every time GAssertMRs evaluates the fitness of an individual it constructs the full Boolean expression $O_f \ [operator] \ F(O_s, I_s, I_f)$.

4.2 Co-Evolutionary Algorithm of GAssertMRs

At the core of both GAssert and GAssertMRs is an evolutionary algorithm that evolves a population P_i to P_{i+1} as follows: (1) **Selection**: it selects from P_i two individuals (parents) by means of a fitness function that rewards fitter solutions. (2) **Reproduction**: it combines the genetic material of the parents (portions of assertions) obtaining two individuals (offspring) by means of crossover operators. (3) **Mutation**: it mutates with a certain probability the offspring by means of mutation operators, and adds the resulting individual to P_{i+1}. The evolutionary algorithm repeats this process until P_{i+1} reaches the predefined size.

GAssert implements a co-evolutionary algorithm that evolves two populations of assertions in parallel, with three competing objectives: (i) minimizing the number of false positives, (ii) minimizing the number of false negatives, (iii) minimizing the size of the assertion. The fitness function used in the first population (ϕ_{FP}) rewards solutions with fewer false positives, while the function of the second population (ϕ_{FN}) those with fewer false negatives. Both populations consider the remaining objectives only in tie cases. Periodically, the two populations exchange their best individuals to provide good genetic material to improve the secondary objectives.

Fitness Functions. Let $FP(\alpha)$ denote the false positive rate of α, and $FN(\alpha)$ denote the false negative rate of α. Both GAssert and GAssertMRs define their multi-objective fitness functions ϕ_{FP} and ϕ_{FN} using the concept of *dominance* ($\prec$) [17] among individuals:

Definition 3. ***FP-fitness (ϕ_{FP}).*** *Given two assertions α_1 and α_2, α_1 **dominates**$_{FP}$ α_2 ($\alpha_1 \prec_{FP} \alpha_2$) if any of the following conditions is satisfied:*

- $FP(\alpha_1) < FP(\alpha_2)$
- $FP(\alpha_1) = FP(\alpha_2) \ \wedge \ FN(\alpha_1) < FN(\alpha_2)$
- $FP(\alpha_1) = FP(\alpha_2) \ \wedge \ FN(\alpha_1) = FN(\alpha_2)$ $\wedge \ size(\alpha_1) < size(\alpha_2)$

Definition 4. ***FN-fitness (ϕ_{FN}).*** *Given two assertions α_1 and α_2, α_1 **dominates**$_{FN}$ α_2 ($\alpha_1 \prec_{FN} \alpha_2$) if any of the following conditions is satisfied:*

- $FN(\alpha_1) < FN(\alpha_2)$
- $FN(\alpha_1) = FN(\alpha_2) \ \wedge \ FP(\alpha_1) < FP(\alpha_2)$
- $FN(\alpha_1) = FN(\alpha_2) \ \wedge \ FP(\alpha_1) = FP(\alpha_2)$ $\wedge \ size(\alpha_1) < size(\alpha_2)$

In tie cases, $FP(\alpha_1) = FP(\alpha_2)$ and $FN(\alpha_1) = FN(\alpha_2)$, both functions favor smaller assertions. This is because smaller assertions are generally easier to understand.

We now describe the selection, reproduction and mutation steps.

Selection. GAssertMRs implements two different selection criteria, tournament and best-match selection, and chooses between them with a given probability.

Tournament Selection [29] is one of the most popular GP selection criterion [45]. The idea is to select the parents by running two "tournaments" among K randomly-chosen individuals. The winner of each tournament (the one with the highest fitness) will be selected [29]. Following the authors of GAssert, we chose $K = 2$, which is known to mitigate the *local optima problem* [45].

Best-match Selection is a new parent selection criterion presented by the authors of GAssert. This criterion selects the first parent randomly, and selects the second parent according to its "complementarity" with respect to the first parent. The concept of "complementarity" is defined by the correct and incorrect executions that an assertion *covers*. An assertion covers a correct execution if it evaluates to true on that execution, while it covers an incorrect execution if it evaluates to false in that execution. The best match selection criterion assigns to each assertion a "weight" that expresses the number of executions that the assertion covers, but are not covered by the first parent. It then selects the second parent using a *weighted random selection*, where assertions with a higher weight are more likely to be selected. Intuitively, this criterion selects with higher probability two parents that "complement" each other in terms of maximizing the collective number of covered correct and incorrect executions. For the population with fitness function ϕ_{FP}, GAssertMRs computes the weights based on the correct executions, while it considers incorrect executions for ϕ_{FN}.

Reproduction exchanges genetic material between two parents producing two offspring, which GAssertMRs mutates (with a given probability). GAssertMRs relies on the canonical tree-based crossover. Given two parents, it selects a random crossover point in each parent, and creates two offspring by swapping the subtrees rooted at each point in the corresponding tree [24].

Mutation The mutation step in the evolutionary algorithm of GAssertMRs relies on three tree-based mutation operators (which are chosen randomly with a given probability).

Node Mutation changes a single node in the tree [12]. It takes as input a tree and one of its nodes n, and returns a new tree obtained by replacing the node n with a new node (chosen randomly).

Subtree Mutation replaces a subtree in the tree [12]. It takes as input a tree and one of its nodes n, and returns a new tree obtained by substituting the subtree rooted at n with another randomly generated subtree.

Constant value mutation changes the value of a single constant within the tree. It takes a tree as its only input, and returns a new tree obtained by randomly selecting a numeric constant node and adding a random number, chosen from $\{-\Delta, \Delta\}$, to its value. Here, Δ should be relatively small (we use 0.1 in our experiments) so that the constant values change in small increments. This mutation operator is not used in the original version of GAssert for assertion oracles. We included this operator because in our experience constants plays a crucial role in MRs.

Note that the original version of GAssert used additional crossover and mutation operators that are inadequate for MRs, and thus are not used by GAssertMRs.

4.3 Test Cases Classification

Just like its predecessor, GAssertMRs requires a dataset of *correct* and *incorrect* behaviours, which are used to calculate the number of false positives and false negatives of a given oracle. This dataset can be used for evaluating the oracles and defining fitness functions that guide the evolution towards oracles with fewer FPs and FNs.

Generally, a *correct* classification should indicate with high certainty that the test output is associated with a correct behaviour. On the other hand, an *incorrect* classification should indicate with high certainty that the output is associated with a failure.

In the context of black-box testing, the elements of this dataset are tests, which include: (1) the test case inputs, (2) the execution outputs, and (3) the classification (*correct* or *incorrect*). Furthermore, for oracles consisting of MRs, these tests should actually be metamorphic tests, i.e., they should include the inputs and outputs from both the source and the follow-up test case executions, as well as their *correct* or *incorrect* classification.

We rely on mutation testing to produce possibly incorrect executions. Hence, all of the test executions from the original system are classified as *correct*, while the test executions from mutants that have an output different from the original for the same test case are classified as *incorrect*. The rest of the executions are left *unclassified* and are removed from the dataset used for training.

For our elevation case study, we define additional constraints to classify an execution from a mutant as *incorrect*. These constraints define a minimum threshold for the difference between the test execution outputs, so that mutant executions with small differences from the original system are not considered *incorrect*. We do this to avoid cases where output measures are so close that a test oracle cannot be reasonably expected to distinguish between them. After consulting with domain experts, we selected the following thresholds for each of our metrics. For the **AWT**, at least 20% increase AND at least 6 seconds increase. For **TD** at least 10% increase AND at least 5 floors increase. Lastly, for **TM** at least 10% increase AND at least 4 movements increase.

There is an additional issue when classifying metamorphic tests from mutants. Each metamorphic test contains a source test case and a follow-up test case, each of which will have its own classification. In this case, we rely on the domain-specific semantics of the QoS metrics being used in order to determine the classification for a test case pair.

For all the 3 QoS metrics that we use in the elevation case study, an *incorrect* classification is always issued when their values are too high. Considering this, we classify the metamorphic tests depending on the relational operator being used for the MRs:

When the source test case is *unclassified* and the follow-up test case is *incorrect*, we classify the metamorphic test as *incorrect* for the MRs with the $<$ or $\leq$ operators. This is because those classifications can be interpreted as the QoS metric value being too high in the follow-up test case, but not in the source test case. Because of this, assertions of the type $O_f \leq F(O_s, I_s, I_f)$ can be expected to identify that O_f is too high. Conversely, for the cases where the source is *incorrect* and the follow-up is *unclassified*, the MRs with $>$ or $\geq$ metrics are classified as *incorrect*. For QoS metrics that are better when they have a higher value, the opposite classification would be applied. For the rest of the cases, the metamorphic tests are considered *unclassified*.

5 EMPIRICAL EVALUATION

To evaluate the proposed approach, we developed a prototype implementation of GAssertMRs, on top of the original implementation of GAssert. We empirically assessed the effectiveness of the generated MRs in detecting failures in the Conventional Group Control (CGC) elevator dispatching algorithm from Orona, using a template configuration from a real building with 10 floors and up to 6 elevators.

5.1 Research Questions

Our experiments aim to answer the following four Research Questions (RQs):

- **RQ1** Is GAssertMRs effective at generating metamorphic relations?
- **RQ2** How does GAssertMRs compare with unguided search for generating metamorphic relations?
- **RQ3** How does GAssertMRs perform when comparing with human-defined metamorphic relations?
- **RQ4** How do generated metamorphic relations compare with similarly generated regular assertions?

The effectiveness of GAssertMRs (RQ1) is determined by its capability to generate metamorphic oracles with (ideally) no false positives and few false negatives. RQ2 checks whether the fitness functions provide useful guidance to generate better MRs. RQ3 compares the effectiveness of the MRs generated automatically by our tool with manually developed MRs, to understand the extent to which GAssertMRs has human-comparable capabilities. RQ4 compares the effectiveness of the automatically generated MRs with regular assertions generated automatically by a modified version of GAssertMRs, which makes use of individual, instead of pairs, of test cases. As for the template for the regular assertions, we use: $M \leq F(I)$, where M is a QoS measure for the selected metric, and $F(I)$ is a function over the inputs of the test case. Just like in GAssertMRs, only the numeric expression $F(I)$ is generated by the evolutionary algorithm. Furthermore, we also defined a mode for "free form" assertions, where an entire Boolean expression can be generated by the tool, similarly to GAssert [40]. For the Boolean expressions, in addition to the regular elements in GAssertMRs expressions, we enable Boolean literals and the following additional operators: $=$, $\neq$, $>$, $<$, $\geq$, $\leq$, AND, OR, IMPLIES and IFF.

5.2 Experimental Setup

Test cases and mutants. We employed mutation testing for evaluating the proposed approach. We used the same set of mutants and test cases that were used for our previous work where the same type of MRs were developed manually [5]. Tests were executed in a domain-specific simulator, which is used by Orona for performing analysis and verification tasks. The test cases were generated for 3 different MRIPs, which we describe below.

The 89 different mutants in the dataset were generated by seeding faults based on traditional arithmetic, logical and relational operator mutations [1]. The faults were seeded manually by a domain expert into the source code for the elevator dispatcher, which is written in the C programming language.

As for the test cases, this dataset contains 1,340 different test cases: 140 source test cases, 420 follow-up test cases for MRIP1, 360 follow-up test cases for MRIP2, and 420 follow-up test cases for MRIP3. Each of these test cases were executed on the original system and the 89 mutants, resulting in a total of 120,600 executions.

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.
- **MRIP2:** ***Additional elevators***. The number of available elevators is increased, without changing the initial positions of the originally available elevators.
- **MRIP3:** ***Initial position change***. The initial positions of all the elevators are shuffled, without changing the number of available elevators.

Initial oracles. In order to generate the initial population for GAssertMRs, we define a simple initial expression for each configuration. For the MR oracles, the atomic numeric expression M_s was used for all cases as right hand side of Equation (MR Template), where M is the QoS metric used in the configuration. For instance, a configuration with the AWT metric as M and the $\leq$ operator would start with $AWT_f \leq AWT_s$ as the initial assertion. The rationale for this is that M_s is an element which is expected to always be present on the right side of the MR for any non-trivial case. For regular assertion oracles, we just used the constant 0 as the initial right-side expression, resulting in initial assertions such as $AWT \leq 0$. Finally, for regular assertions that do not have a fixed structure, we used *true* as the initial assertion. Half of the initial population for GAssertMRs is generated by applying random mutations to the initial assertion, and the remaining half is generated randomly.

Results validation. In order to validate the ability to generalize from the training dataset, we employed 10-fold cross-validation for all of the evaluations. The dataset was divided by tests (test case pairs for MRs or individual test cases for the regular assertions), which ensures that the training and testing datasets contain the same proportion of *correct* and *incorrect* execution data. Each different configuration was executed 10 times, with each execution using a different subsample as the testing set, and the remaining 9 subsamples as the training set. We used the same partitions for all of the MR generation approaches, but regular assertions used different partitions due to the samples being individual test cases rather than test case pairs. For the manually developed MRs, the training set

Table 2: GAssertMRs Configuration Parameters.

Parameter Description	Value	Parameter Description	Value
bound on the size of the assertions	32	time budget (minutes)	15
size of each of the populations (N)	1,000	prob. of crossover	90%
minimum number of generations	100	prob. of mutation	30%
maximum number of generations	10,000	prob. of tournament selection	50%
frequency of elitism (every X gen)	1	prob. of best-match selection	50%
frequency of migration (every X gen)	10	prob. constant mutation min	5%
number of assertions for elitism	10	prob. constant mutation max	50%
number of assertions to migrate (M)	160	increase prob. const. mut. every gen.	0.45%

was not needed, we just used the testing sets for the evaluation and comparison with GAssertMRs. Furthermore, in order to account for the stochastic nature of GAssertMRs, we repeated each of the experiments 12 times with different random seeds. In total, each GAssertMRs configuration was executed $10 \times 12 = 120$ times.

Evaluation metrics. We evaluated the proposed approaches with the following four metrics.

FP and **FN** refer to the percentage of false positives and false negatives based on the classification provided to the evolutionary algorithm. Hence, these are the final fitness results obtained by the algorithm. Note that while FP directly represents the percentage of actual false positives, there might be more detected faults than FN suggests, since some of the mutant test cases were deemed *unclassified*, rather than being classified as *incorrect* (see Section 4.3). Hence, they are not counted either as true positives or as false negatives.

Detected failures (**DF**) refers to the number of failing verdicts across mutant test executions. On the other hand, the mutation score (**MS**) is the percentage of mutants detected out of the 89 mutants in the dataset. A mutant is considered detected by an assertion if the assertion returns false in at least one of the executions.

In the cases where a given assertion caused a false positive in a test, the results from the rest of the mutant executions for that test are not evaluated, i.e., the assertion is considered to detect 0 failures in that test. This means that all the instances classified as *incorrect* for that test will count as FNs, and the results from the test will not increase DF and MS. Furthermore, in the case of an error occurring during the assertion evaluation (e.g. division by 0), a failing verdict is assigned.

Configuration. Table 2 shows the parameter values used by GAssertMRs and by its Regular Assertions variant used for RQ4. The Unguided variant used for RQ2 also used the same parameter values, except for the parent selection, which was 100% random instead, and the elitism and migration parameters, which were irrelevant in this case because elitism and migration were disabled.

These parameter values were taken from the previous GAssert experiments [40]. One minor tweak that has been made is using a dynamic value for the probabilities of either using or altering a constant during mutation. This dynamic value increases linearly with the generation number, starting from 5%, up to 50% at generation 100. The reason for the linear increase is to allow more micro-optimizations of constant values towards the later generations.

We configured GAssertMRs with each of the 3 different MRIPs used in our case study, combined with the 3 QoS metrics extracted, and 2 relational operators ($\geq$ and $\leq$). Hence, in total there are 18 different configurations of GAssertMRs (3 MRIPs $\times$ 3 metrics $\times$ 2 operators). When generating regular assertions in free form mode, we considered different configurations where each of the QoS metrics is used individually, as well as an additional configuration where all the available QoS metrics can be used in the same assertion.

We executed the 12 random seeds of each configuration across 4 identical virtual machines (3 random seeds per machine) running on a server. Each virtual machine had an 8-core 3.2GHz CPU and 16 GB of RAM assigned. The total machine time of the experiments was 540 hours for GAssertMRs (12 seeds $\times$ 10 folds $\times$ 18 configurations), another 540 hours for the Unguided variant, and 210 hours for the Regular Assertions variant (12 seeds $\times$ 10 folds $\times$ 7 configurations).

5.3 Results

Table 3 shows the evaluation results for GAssertMRs, as well as the ones for the Unguided and Manual MRs baselines. The reported numbers are the median obtained on the testing dataset across each fold and random seed. For fair comparison, the results of Manual MRs were also calculated separately over the testing subsamples of each of the 10 folds and the median results are reported.

Table 4 shows the evaluation results for the Regular Assertions baseline, where every evaluation metric reported is again the median of the results obtained on the testing dataset across each of the folds and random seeds.

Figure 1 shows the box plots of the FNs (percentage, from 0.00 to 1.00). Each of these box plots contains 120 data points for each tool and configuration (12 random seeds $\times$ 10 folds).

RQ1: Effectiveness of GAssertMRs. Column GAssertMRs of Table 3 shows the median results obtained by our proposed approach for each of the possible configurations. The results show that GAssertMRs is able to generate non-trivial MRs which have very few FPs and are capable of detecting some failures in the mutant test cases. While the median of FPs is 0 and the mean was always less than 1, there were actually some FPs in the testing dataset.

On the other hand, we observe a very large percentage of FNs in many configurations, which may indicate that the fitness function is difficult to satisfy. Furthermore, the results show significantly different performance across the different configurations, indicating that some MRIP, QoS metric and operator combinations might be easier to generate or have more potential effectiveness.

It is important to note that the results shown in Table 3 are obtained with the testing dataset, which represents only 10% of the test cases, and therefore the mutation score is expected to be low. For reference, the Manual MRs obtain an average mutation score of over 30% with the full test suite, with one of the configurations achieving a mutation score of over 80%. Furthermore, we do not aggregate the results from multiple random seeds, folds, or configurations.

RQ2: Comparison with Unguided Search. Comparing the results by GAssertMRs and Unguided in Table 3, we can see that the former tends to obtain significantly better results for FN, DF and MS in many cases, and similar results in the worst cases. There is only a single instance where Unguided detected more failures, and 2 instances where it obtained a better mutation score, and the

Table 3: Evaluation Results for RQ1, RQ2 and RQ3 (median).

MRIP	Metric	Operator	GAssertMRs (RQ1) FP	FN	DF	MS	Unguided (RQ2) FP	FN	DF	MS	Manual MRs (RQ3) FP	FN	DF	MS
MRIP1	AWT	≥	0.00%	95.99%	16.5	11.80%	0.00%	97.22%	9.0	7.87%	0.00%	92.13%	13.0	11.24%
		≤	0.00%	88.20%	28.0	14.61%	0.00%	90.00%	18.0	13.48%	0.00%	90.03%	14.0	14.61%
	TD	≥	0.00%	89.09%	12.0	12.36%	0.00%	89.77%	10.5	11.24%	0.00%	90.39%	12.5	13.48%
		≤	0.00%	92.50%	**30.5**	13.48%	0.00%	94.19%	**19.0**	8.99%	0.00%	98.93%	**4.5**	5.06%
	TM	≥	0.00%	**66.67%**	**20.0**	16.85%	0.00%	**75.00%**	**12.0**	11.24%	0.00%	**88.13%**	**7.5**	7.87%
		≤	0.00%	93.54%	9.0	4.49%	0.00%	92.00%	5.0	2.25%	0.00%	100.00%	0.0	0.00%
MRIP2	AWT	≥	0.00%	93.56%	**11.0**	**7.87%**	0.00%	96.19%	4.0	4.49%	0.00%	100.00%	**0.0**	**0.00%**
		≤	0.00%	65.20%	74.0	19.10%	0.00%	62.77%	74.0	19.10%	0.00%	64.72%	74.0	18.54%
	TD	≥	0.00%	96.70%	9.0	**8.43%**	0.00%	98.59%	4.0	3.37%	0.00%	100.00%	0.5	**0.56%**
		≤	0.00%	**82.29%**	**47.5**	12.36%	0.00%	84.52%	51.0	15.73%	0.00%	**96.57%**	**8.5**	9.55%
	TM	≥	0.00%	100.00%	2.0	1.12%	0.00%	100.00%	0.0	0.00%	0.00%	96.56%	1.5	1.12%
		≤	0.00%	**76.92%**	**22.0**	8.99%	0.00%	**83.87%**	**15.0**	5.62%	0.00%	**100.00%**	**2.0**	2.25%
MRIP3	AWT	≥	0.00%	90.57%	**28.5**	**13.48%**	0.00%	94.03%	**18.0**	8.43%	0.00%	98.51%	**1.5**	**1.12%**
		≤	0.00%	**85.85%**	**50.0**	**19.10%**	0.00%	88.62%	42.0	17.98%	0.00%	**100.00%**	**0.0**	**0.00%**
	TD	≥	0.00%	89.01%	**58.0**	**17.98%**	0.00%	94.19%	**24.0**	11.24%	0.00%	98.93%	**1.5**	**1.69%**
		≤	0.00%	**89.35%**	**30.0**	**12.36%**	0.00%	89.65%	30.0	12.92%	0.00%	**99.57%**	**0.5**	**0.56%**
	TM	≥	0.00%	**80.00%**	**16.0**	7.87%	0.00%	**90.40%**	**5.0**	2.25%	0.00%	**100.00%**	**0.5**	0.56%
		≤	0.00%	**51.92%**	**19.0**	2.25%	0.00%	54.17%	19.0	2.25%	0.00%	**100.00%**	**0.0**	0.00%

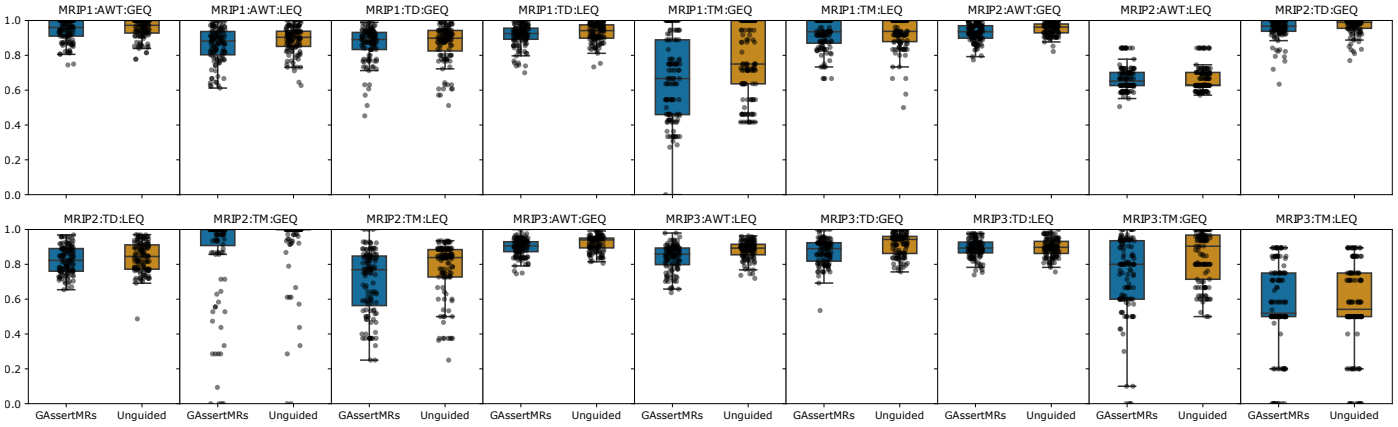

Figure 1: False Negatives for GAssertMRs and Unguided (RQ1 and RQ2)

difference is minimal in most cases. Figure 1 provides a detailed visualization of the FNs of GAssertMRs vs Unguided.

These results were further corroborated by means of statistical tests. After obtaining the experimental results, we applied the Shapiro-Wilk test to assess how the data was distributed. Since the data was not normally distributed, we employed the Mann-Whitney U-test to assess the statistical significance of the differences. The statistical significance threshold for the p-value was set to 0.05. In addition, we assessed the effect size by employing the Vargha and Delaney $\hat{A}_{12}$ metric. As suggested in [32], we categorized the difference existing between GAssertMRs and Unguided as *negligible* if $d < 0.147$, as *small* if $d < 0.33$, as *medium* if $d < 0.474$ and as *large* if $d >= 0.474$, where $d = 2|\hat{A}_{12} - 0.5|$.

For the mutation score, in 16 out of 18 combinations, the results were in favor of GAssertMRs. Out of these 16 cases, the difference was large twice, medium twice, small in 9 cases and negligible in 3 cases. Out of these 16 cases, 11 were statistically significant. Conversely, for 2 out of 18 combinations where Unguided performed better, only in one of the cases Unguided did it with statistical significance, and the difference was small. For the FNs, the $\hat{A}_{12}$ was in favor of GAssertMRs with respect to Unguided in 17 out of 18 cases, with 9 of them having statistical significance. For 7 cases out of the 17, the difference was negligible, for the other 7 small and for the remaining 3 medium. For the only case where the $\hat{A}_{12}$ value was not in favor of GAssertMRs, the difference was negligible and the p-value was 0.91, which means that the results were not distinguishable. As for the $\hat{A}_{12}$ value for FPs, Unguided showed better results than GAssertMRs in all cases, but the results were statistically significant only in 6 out of 18 cases. The difference between the FPs of both techniques were negligible in 14 of the cases and small in the remaining 4 cases. This overall means that GAssertMRs has a slightly higher tendency to get FPs than Unguided, also a significantly superior FN reduction capability.

Table 4: Evaluation Results for RQ4 (median).

Metric	Operator	GAssert (RQ4) FP	FN	DF	MS
All	Free form	0.00%	97.40%	34.0	14.61%
AWT	≤	0.00%	91.33%	83.0	21.35%
	Free form	0.00%	96.08%	42.5	16.85%
TD	≤	0.00%	95.91%	33.5	10.11%
	Free form	0.00%	96.64%	28.0	8.99%
TM	≤	0.00%	92.45%	8.5	3.37%
	Free form	0.00%	93.94%	8.5	2.25%

RQ3: Comparison with Manual MRs. The original manually generated MRs [5] were implemented in Python, following the same QoS MR template as the MRs generated by GAssertMRs. We use new versions of these MRs that solve incompatibilities with GAssertMRs, which ensures that all them could be generated by it. These new versions of the MRs either match or outperform the older versions in both failure detection ratio and mutation score. Furthermore, they also have 0 false positives in the full test case dataset.

The results in Table 3 show that Manual MRs outperform the failure detection counts or mutation scores of GAssertMRs in just one instance each. Furthermore, Manual MRs only outperform GAssertMRs by a narrow margin in both of these cases. On the other hand, there are several configurations where GAssertMRs outperforms the results of Manual MRs by a very significant margin. This indicates that GAssertMRs might be more effective at detecting a higher number of diverse failures. On the other hand, Manual MRs have other advantages over automatically generated MRs, such as: (1) exactly 0 FPs on all evaluation tests; (2) lower complexity (max 6 for Manual vs 32 for GAssertMRs).

RQ4: Comparison with Regular Assertions. We compare the GAssertMRs results from Table 3 with the results from its Regular Assertions variant in Table 4. Here, we can see that there are several configurations where GAssertMRs outperforms the Regular Assertions variant in terms of DF and MS. We do not plot the distributions of the FNs, since they are not directly comparable with the MRs.

In fact, for oracles based on the TD and TM QoS metrics, MRs achieve better results (more failures detected and higher mutation scores) with the best configurations of GAssertMRs than with the best configurations of the Regular Assertions variant. However, the configuration for Regular Assertions with the best results (AWT with the ≤ operator) outperforms all of the results obtained by MRs in both failure detection count and mutation score.

Moreover, as with the automatically generated MRs, the number of FPs is not always 0 in all the configurations, but the average is still less than 1, and the median is 0, as shown in Table 4.

It is important to note that the number of available tests is not equal for MRs and Regular Assertions, since MRs employ test case pairs for a specific MRIPs, whereas Regular Assertions use the individual results from all the test case executions. In terms of individual test cases, the configurations for MRIP1 and MRIP3 could use 140+420 = 560 test cases, whereas the configurations for MRIP2 could use 140 + 360 = 500 test cases. In comparison, each Regular Assertions run could use all of the 1340 different test cases.

The fact that each configuration for MRs can only use a smaller subset of the test cases can be considered an inherent disadvantage of using metamorphic oracles. On the other hand, this allows generating more different MR oracles, because we can (and should) generate different output relations for every MRIP. However, more configurations also means that the effort needed for generating these oracles is higher. In this case, we have 18 configurations (rows in Table 3) for MRs, and 7 configurations (rows in Table 4) for regular assertions, so the cost of generating all the MRs is $\frac{18}{7} \approx 2.57$ times higher than the cost of generating all the regular assertions.

6 LESSONS LEARNED

We now discuss the lessons learned from this experience, and the future work for the industrial adoption of this approach.

Lesson 1 – Automated generation of MRs is feasible with GAssertMRs in practice: The industrial need to identify techniques for automatically inferring metamorphic relations has been acknowledged in a recent industry-relevant study [2]. In this paper we demonstrate the advance over the state-of-the-practice in this direction by extending GAssert [40] to accommodate the needs of generating MRs in an industrial context. Despite its long history, the adoption of metamorphic testing by industrial companies has started in the last few years in companies like Facebook [2], Adobe [44], or Orona [5]. Nevertheless, the definition of MRs in all these contexts has remained manual, relying on the domain knowledge of experts. Our results show that GAssertMRs is competitive with manually-defined MRs, outperforming them in some cases.

Besides its effectiveness, it is important to highlight that GAssertMRs generates MRs automatically. According to the engineers from Orona, GAssertMRs provides significant benefits because it is able to find MRs that are otherwise difficult to be found by engineers. In addition, elevation experts are not experienced in software testing, which significantly increases the manual cost of defining MRs.

The correct and incorrect executions needed by GAssertMRs can be easily obtained. Indeed, manually validated field executions are often available for industrial CPSs, so these can be used as correct executions for GAssertMRs. They can then be mutated to obtain the corresponding incorrect executions and fully automate the MR generation process.

Lesson 2 – Small test suites are sufficient for the generation of MRs in an industrial context: The main difference between software-only applications and CPSs is the added difficulty and cost of executing tests. For complex CPSs it may be infeasible to generate a large dataset of test executions the same way GAssert did for Java programs [40]. A small dataset might result in the evolutionary algorithm generating poor quality MRs. In this paper, we show that GAssertMRs can generate useful MRs with a relatively small set of test executions that were generated at an affordable cost. It is worth noting that simulated test executions might not be an adequate solution for some types of systems, due to excessive simulation costs or lack of accuracy in the simulations.

An alternative approach is to maintain a database of real test executions (with manually identified correct and incorrect behaviours), which can also be used to generate MRs with GAssertMRs.

Lesson 3 – Setting constraints on the generated MRs can make the approach more effective: The current implementation of GAssertMRs relies on numerical inputs and outputs, since the expression trees it generates only support this type of variables. Although it would be possible to extend the current expression tree and evaluator in order to support additional data types and operations, this has not been considered necessary within the context of the current case study. For CPSs, the input/output domains consist of numerical variables in most cases [3]. For other cases, which occurred also in our elevation case study, it is possible to write user-defined functions that transform complex inputs into numerical variables. Although defining these functions requires some effort, this approach greatly reduces the cost of running the tool in terms of execution time (simpler expressions result in a smaller search space), and also results in more familiar MRs. Moreover, our experiments use a specific MR template, although it would be possible to make the tool generate arbitrary expressions with Boolean and numeric variables. The reasons for enforcing such a template are (1) reducing the search space for GAssertMRs, and (2) making it easier to interpret the generated MRs. We believe that the use of the proposed template would make sense for many QoS or performance CPS testing contexts.

Future work – Further steps required for adoption by industrial practitioners: In order to encourage the adoption of GAssertMRs, further steps need to be taken. The current version of GAssertMRs requires metamorphic input relations (or equivalently, the transformations to apply to the inputs) and test cases. The automated generation of input relations and test cases is out of the scope of the current implementation. The development of a Domain Specific Language (DSL) to support engineers in the definition of input relations is envisioned to help the easy transfer of the tool to practitioners. Given a user-defined input relation, it should be possible for a tool to automatically generate new source and follow-up test cases, similarly to how OASIs [18, 40] does for GAssert. Mutant generation could also be automated, although executing all test cases in all the new mutants may be too costly in the context of CPSs. For an even more automated approach, it would also be possible for the evolutionary algorithm to generate and improve both the input and the output relations. The availability of a generic tool which implements this approach would greatly encourage the adoption by industrial practitioners.

7 THREATS TO VALIDITY

External validity. The main external validity threat relates to the generalization of our results. Although we are reporting the results of a single case study, it is important to highlight that it is a real industrial case study that involve a complex CPS. Nevertheless, we acknowledge that our results might not be representative of other CPSs from other domains.

Internal validity. Another potential threat to the validity of our empirical evaluation is the use of mutation testing, which might have introduced a bias in our results. Mutation testing introduces a bias when the mutants are too few or there are many equivalent mutants [31]. To mitigate this threat, we generated an amount of mutants comparable to other related works that use simulation-based mutation testing [4, 27, 28]. Furthermore, we also checked these mutants to identify and filter out equivalent mutants, as recommended by Papadakis et al. [31].

Construct validity. To ensure that our evaluation supports our conclusions, we generate multiple configurations of GAssertMRs, and compare the results with multiple baselines.

8 RELATED WORK

Metamorphic testing for CPSs. Metamorphic testing has already been used for testing CPSs in order to mitigate the oracle problem. Lindvall et al. combined metamorphic testing and model based testing approaches in order to perform simulation-based testing of autonomous drones [26]. Other techniques have also applied metamorphic testing to verify autonomous self-driving cars [42, 47]. However, those works do not focus on the generation of MRs, but the application of them.

MR generation. Several semi-automatic and automatic MR identification approaches have already been explored in the existing literature. Many publications have suggested reusing abstract patterns in the input or the output relations in order to derive concrete MRs for a particular system [34, 48]. In this work, we reuse the same, manually defined, Metamorphic Relation Input Patterns (MRIPs) to generate different MRs. Troya et al. proposed an approach for automatically inferring MRs for model transformations based on a catalogue of metamorphic relation patterns which may apply to any model transformation [43]. Zhang et al. proposed a machine learning approach which infers MRs with polynomial input and output relations based on program executions [46]. In contrast, GAssertMRs generates MRs following a template for output relations specifically designed for QoS metrics.

9 CONCLUSION

In this paper, we investigate the automated generation of Metamorphic Relations (MRs) in an industrial setting. Towards this goal, we implemented GAssertMRs, a tool to automatically generate MRs. GAssertMRs implements an evolutionary algorithm which attempts to minimize the false positive and false negative rates of the generated MRs. We define a generic output relation template for quality of service testing and various domain-specific input relations for elevators, and we employ random test generation and seeded faults in order to generate a dataset of *correct* and *incorrect* test executions.

Our study shows that the generated metamorphic relations can be effective in the context of an industrial CPS, enabling the automation of a significant part of the testing process at an affordable cost. Future work will explore the complete automation of the metamorphic relations generation, including the definition of the MRIPs and the mutant generation. On the other hand, GAssertMRs will also have to be evaluated with other CPSs in order to assess its usefulness in different application domains.

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 H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

REFERENCES

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

[2] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, et al. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *Proceedings of the 43rd International Conference on Software Engineering*.

[3] Rajeev Alur. 2015. *Principles of cyber-physical systems.* MIT Press.

[4] 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.

[5] 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, IEEE, 104–114.

[6] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Experimental Data and Results. https://drive.google.com/drive/folders/18XwGyV1tjFkqSvpT-gs-i35uJEnTlEsI. Last access: Jul 2021.

[7] Thomas Back. 1996. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press.

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

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

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

[11] Mohamed Boussaa, Olivier Barais, Gerson Sunyé, and Benoit Baudry. 2020. Leveraging metamorphic testing to automatically detect inconsistencies in code generator families. *Software Testing, Verification and Reliability* 30, 1 (2020), e1721. https://doi.org/10.1002/stvr.1721 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1721 e1721 stvr.1721.

[12] Markus F Brameier and Wolfgang Banzhaf. 2007. A Comparison with Tree-Based Genetic Programming. *Linear Genetic Programming* (2007), 173–192.

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

[14] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *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.

[15] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. 2004. Case Studies on the Selection of Useful Relations in Metamorphic Testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. 569–583.

[16] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[18] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 247–258.

[19] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2019. An Empirical Validation of Oracle Improvement. *IEEE Transactions on Software Engineering* (2019).

[20] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[21] O. Johnston, D. Jarman, J. Berry, Z. Q. Zhou, and T. Y. Chen. 2019. Metamorphic Relations for Detection of Performance Anomalies. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. 63–69.

[22] 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.

[23] Siddhartha Kumar Khaitan and James D McCalley. 2014. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal* 9, 2 (2014), 350–365.

[24] John R Koza and John R Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Vol. 1. MIT press.

[25] Edward Ashford Lee and Sanjit A Seshia. 2016. *Introduction to embedded systems: A cyber-physical systems approach.* Mit Press.

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

[27] Bing Liu, Shiva Nejati, Lionel C Briand, et al. 2017. 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)*. IEEE, 359–370.

[28] 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.

[29] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems* 9, 3 (1995), 193–212.

[30] Orona. 2021. Orona Group. https://www.orona-group.com/. Last access: Jan 2021.

[31] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 936–946.

[32] 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*. 1–51.

[33] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (Sept 2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[34] S. Segura, J.A. Parejo, J. Troya, and A. Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering* 44, 11 (Nov 2018), 1083–1099. https://doi.org/10.1109/TSE.2017.2764464

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

[36] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés. 2017. 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)*. 7–10.

[37] Sergio Segura, Javier Troya, Amador Durán, and Antonio Ruiz-Cortés. 2018. Performance metamorphic testing: A Proof of concept. *Information and Software Technology* 98 (2018), 1 – 4. https://doi.org/10.1016/j.infsof.2018.01.013

[38] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. 1996. Multi-objective optimization by genetic algorithms: A review. In *Proceedings of IEEE international conference on evolutionary computation*. IEEE, 517–522.

[39] Valerio Terragni, Gunel Jahangirova, Mauro Pezzè, and Paolo Tonella. 2021. Improving Assertion Oracles with Evolutionary Computation. In *Proceedings of the Genetic and Evolutionary Computation Conference, Hot Off the Press track (GECCO 2021)*.

[40] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Virtual Event, USA, 8–13 November, 2020*. 1178–1189. https://doi.org/10.1145/3368089.3409758

[41] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2021. GAssert: A Fully Automated Tool to Improve Assertion Oracles. In *Proceedings of the 43nd IEEE/ACM International Conference on Software Engineering, Demonstration Track (ICSE 2021)*.

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

[43] J. Troya, S. Segura, and A. Ruiz-Cortés. 2018. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software* 136 (2018), 188 – 208. https://doi.org/10.1016/j.jss.2017.05.043

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

[45] Darrell Whitley. 1994. A Genetic Algorithm Tutorial. *Statistics and Computing* 4, 2 (1994), 65–85.

[46] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. 2014. Search-based Inference of Polynomial Metamorphic Relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 701–712. https://doi.org/10.1145/2642937.2642994

[47] Zhi Q Zhou and Liqun Sun. 2019. Metamorphic testing of driverless cars. (2019).

[48] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey. 2018. Metamorphic Relations for Enhancing System Understanding and Use. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2876433