This is an Accepted Manuscript version of the following article, accepted for publication in:

J. Ayerdi, V. Terragni, G. Jahangirova, A. Arrieta and P. Tonella, "GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming," in *IEEE Transactions on Software Engineering*.

GENMORPH: Automatically Generating Metamorphic Relations via Genetic Programming

Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella

Abstract—Metamorphic testing is a popular approach that aims to alleviate the oracle problem in software testing. At the core of this approach are Metamorphic Relations (MRs), specifying properties that hold among multiple test inputs and corresponding outputs. Deriving MRs is mostly a manual activity, since their automated generation is a challenging and largely unexplored problem. This paper presents GENMORPH, a technique to automatically generate MRs for Java methods that involve inputs and outputs that are boolean, numerical, or ordered sequences. GENMORPH uses an evolutionary algorithm to search for *effective* test oracles, i.e., oracles that trigger no false alarms and expose software faults in the method under test. The proposed search algorithm is guided by two fitness functions that measure the number of false alarms and the number of missed faults for the generated MRs. Our results show that GENMORPH generates effective MRs for 18 out of 23 methods (mutation score >20%). Furthermore, it can increase RANDOOP's fault detection capability in 7 out of 23 methods, and EVOSUITE's in 14 out of 23 methods. When compared with AUTOMR, a state-of-the-art MR generator, GENMORPH also outperformed its fault detection capability in 9 out of 10 methods.

Index Terms—metamorphic testing, oracle problem, metamorphic relations, genetic programming, mutation analysis

✦

1 INTRODUCTION

The increasing complexity of software systems makes software test automation both increasingly important and challenging. One of the main issues in this domain is the automation of the test oracle problem, the problem of distinguishing between correct and incorrect test executions [1]. While recent years have witnessed significant advances in automated test input generation [2], [3], [4], [5], [6], the oracle problem remains one of the main bottlenecks to achieve full test automation [7].

Metamorphic Testing (MT) [8] aims at alleviating the oracle problem. It is based on the intuition that even if we cannot automatically determine the correctness of the output for an individual input, it may still be possible to use relations among the expected outputs of multiple related inputs as oracles [9]. Existing research on metamorphic testing has proven that such relations, called **Metamorphic Relations (MRs)**, exist in virtually any non-trivial software system [10].

For example, consider a program `f` that computes the square of a number. We cannot determine the correctness of `f(5)` without knowing that the expected value is 25. However, a mathematical property of the square function is $x^2 = (-x)^2$. This property can be represented as an MR that verifies that the square functions of two numbers (one the negation of the other) are identical. If two inputs violate this MR, we have found a fault in the implementation of `f`.

The key advantage of MT is that a single MR can be applied to multiple test inputs. This is generally not true for canonical test assertions that often predicate on specific test inputs. For example, the test assertion `assertEquals(16,f(4))` is specific to the input 4. Manually deriving assertions for thousands of automatically generated test inputs is infeasible. Indeed, test generators often need many test inputs to be effective [11]. Conversely, a single MR can be applied to different test inputs, as long as they satisfy the input relation.

With the rapid advance of test input generators, the popularity of metamorphic testing has drastically increased [9], [10], [12]. Recently, large companies such as META [13], GOOGLE [14], ADOBE [15], and NASA [16] are leveraging MT for testing their software systems. While the usefulness of MT is well recognized among both researchers and the industry [12], identifying MRs largely remains a costly manual activity that requires domain knowledge [17]. The automated discovery of MRs is an important research topic in MT, as it would reduce the human cost associated with metamorphic testing and enable full test automation [9], [10], [12]. Unfortunately, it is a challenging problem that remains largely unexplored [9], [10], [12], [18], [19].

This paper presents **GENMORPH** (*Generator of Metamorphic Relations*) a technique to automatically discover MRs for Java methods. GENMORPH relies on search-based software engineering (in particular, genetic programming) to explore the space of candidate solutions, driven by a fitness function that rewards MRs with fewer false positives and false negatives. In the context of test oracles [20], *false positives* represent correct program executions in which the oracle fails but should pass, and *false negatives* represent incorrect program executions in which the oracle passes but should fail [20], [21], [22]. Correspondingly, a high-quality MR is one with no false alarms (zero false positives) that is useful to expose software faults (few false negatives).

We evaluated GENMORPH on 23 Java methods from

- J. Ayerdi and A. Arrieta are with the Department of Electrical and Computer Engineering, Mondragon University, Spain.
  E-mail: jayerdi@mondragon.edu
- V. Terragni is with Auckland University, New Zealand
- G. Jahangirova is with the King's College London, United Kingdom
- P. Tonella is with the Università della Svizzera italiana (USI), Switzerland

three different open-source libraries. The results show that GENMORPH generates effective MRs for 18 of these subjects. Furthermore, the generated MRs increase the fault detection capability of automatically generated regression test suites for 14 subjects. Compared with AUTOMR, a state-of-the-art MR generation tool, GENMORPH achieved a higher fault detection capability with 9 out of 10 subject methods. In summary, this paper makes the following contributions:

- It proposes GENMORPH, a novel technique to automatically generate valid and effective MRs for functions with numerical, Boolean, array, List and String inputs and outputs.
- It describes a framework to automatically evaluate MRs with automated test generation and mutation analysis.
- It presents an empirical evaluation of GENMORPH involving 23 functions from three open-source libraries.
- It releases a replication package to facilitate future work [23], including the source code of GENMORPH[1] and the results from our experiments.

2 PROBLEM FORMULATION

This section gives the background of this work and formulates the problem of generating effective MRs. We follow the seminal definition of MRs by Chen et al. [8], [9]:

Definition 1. Let f be a target function or algorithm. A **Metamorphic Relation (MR)** is a necessary property of f over a sequence of inputs $\langle x_1, \cdots, x_n \rangle$ where $n \geq 2$, and their corresponding outputs $\langle f(x_1), \cdots, f(x_n) \rangle$.

In this work, we focus on the subset of MRs that can be represented as a logical implication ($\Rightarrow$) between a relation R_i defined over the sequence of inputs, and a relation R_o defined over the corresponding sequence of outputs. Furthermore, we focus on MRs involving pairs of inputs and corresponding outputs ($n = 2$), which characterize most of the MRs presented in the literature [10].

$$R_i(x_1, x_2) \Rightarrow R_o(f(x_1), f(x_2))$$

Whenever a given input relation $R_i(x_1, x_2)$ holds between the two inputs x_1 and x_2, a corresponding output relation $R_o(f(x_1), f(x_2))$ is expected to hold between the outputs.

Referring to the square function example discussed in the introduction, the presented metamorphic relation is defined as:

$$(x_1 = -x_2) \Rightarrow (f(x_1) = f(x_2))$$

A MR can be used as a **metamorphic test oracle**: an executable Boolean expression that reports an invariant violation if the input relation is satisfied (true), but the output relation is not (false).

Typically, metamorphic test cases are generated by first obtaining an initial test input (by applying any regular test generation strategy), and then applying a transformation to it ($\rightsquigarrow$) such that the input relation will be satisfied. In such cases, the initially generated test input is called the **source test input** (or source test case), and the one derived from it to satisfy the input relation is called the **follow-up test input** (or follow-up test case). Formally, given an MR, we call a

1. https://github.com/jonayerdi/genmorph

(metamorphic) input transformation [24] a transformation of the inputs $x_1 \rightsquigarrow x_2$ that satisfies $R_i(x_1, x_2)$.

Referring to the square function example, an input transformation for its *MR* is $x_1 \rightsquigarrow -1 \cdot x_1$. For example, given the source test $x_1 = 4$, its follow-up is $x_2 = -4$.

Given a single MR, MT can generate an arbitrary number of source test inputs and use the *input transformation* to automatically create the corresponding follow-up test inputs. MT executes the function under test on each pair of inputs and reports an oracle violation if the output relation is false.

Clearly, the effectiveness of metamorphic testing highly depends on the specific MRs that are used. Thus, designing effective MRs is a critical step when applying MT [9], [10], [12]. However, discovering effective MRs is labor-intensive and usually requires advanced domain expertise [8], [9], [10].

Given a function f and its implementation P, GENMORPH aims at automatically generating one or more MRs that are **"effective"** at exposing faults in P. We measure effectiveness in terms of false positives (FP) and false negatives (FN) of a test oracle [20], [21].

Jahangirova et al. [20] define a FP as a correct program execution in which the oracle fails but should pass, and a FN as an incorrect program execution in which the oracle passes but should fail. However, these definitions are ill-suited for MT because metamorphic oracles predicate on multiple test executions. As such, we modify them as follows:

Definition 2. Let P be an implementation of a target function or algorithm f. A **false positive (FP)** of a metamorphic oracle MR is a pair of inputs x_1 and x_2 such that both $P(x_1)$ and $P(x_2)$ are correct and $[R_i(x_1, x_2) \Rightarrow R_o(P(x_1), P(x_2))] =$ false (i.e., $R_i(x_1, x_2) =$ true and $R_o(P(x_1), P(x_2)) =$ false).

Definition 3. Let P be an implementation of a target function or algorithm f. A **false negative (FN)** of a metamorphic oracle MR is a tuple $\langle x_1, x_2, \mu P \rangle$, where x_1 and x_2 is a pair of inputs and μP is an incorrect version of P (e.g., seeded fault) such that $[\mu P(x_1) \neq P(x_1)$ or $\mu P(x_2) \neq P(x_2)]$ and $[[R_i(x_1, x_2) \Rightarrow R_o(\mu P(x_1), \mu P(x_2))] =$ true$]$.

In Definition 2, the condition that both $P(x_1)$ and $P(x_2)$ are correct can be assumed to hold in a regression context, where we are interested in modeling the implemented behavior using a metamorphic oracle that can be adopted later for regression purposes. In Definition 3, the condition $[\mu P(x_1) \neq P(x_1)$ or $\mu P(x_2) \neq P(x_2)]$ checks that executing the mutant μP with at least one of the inputs (x_1 or x_2) corrupts the output value. If not, the metamorphic oracle has no means to expose the seeded fault.

> **Problem Definition:** *Given a function f and its implementation P, automatically generate one or more MRs that have zero false positives and the fewest false negatives.*

There are two important considerations to make: First, differently from program assertions [25] and pre/post conditions [26], a single MR usually does not predicate on all possible program inputs and thus can hardly achieve zero FNs. An MR only predicates on those inputs that satisfy the input relation. For this reason, outputting multiple MRs with different input relations is useful as they might complement each other on the types of faults they can detect. Conversely,
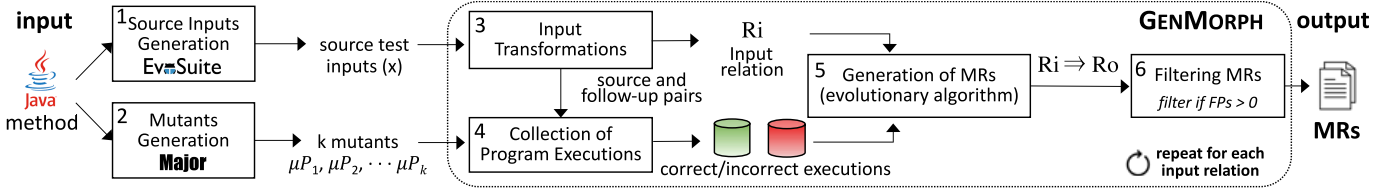
Fig. 1: Logical architecture of GENMORPH for the automated generation of MRs

aiming at zero FPs is highly desirable because of the high cost of manually inspecting false alarms.

Second, same as Terragni et al. [25] and Molina et al. [26], we are considering the *implemented* program behavior for false positives, which might differ from the *intended* one. As such, GENMORPH might need a manual validation of the generated MRs to ensure that they capture the intended program behavior.

3 GENMORPH

GENMORPH takes as inputs the implementation P of a function (method in Java) and a time budget. It explores the space of possible MRs for P, and when the budget expires, it returns the most "effective" MRs explored so far. GENMORPH explores the space of possible MRs with an evolutionary algorithm driven by fitness functions that reward MRs with fewer false positives (FPs) and false negatives (FNs). Figure 1 overviews GENMORPH, which generates MRs in six steps:

1) Source Inputs Generation. GENMORPH needs a set of input pairs (source and follow-up tests) to discover and evaluate MRs. Our current implementation for Java methods employs EVOSUITE [2] to generate a diverse set of source test inputs. EVOSUITE generates test inputs driven by branch coverage, which guarantees that the generated tests cover diverse execution paths of the method under analysis [27]. Since the input relation is still unknown at this stage, GEN-MORPH uses these generated tests as source test inputs only. Note that one could also add manually-written test inputs.

2) Mutants Generation. To obtain a dataset of *incorrect* test executions, GENMORPH executes the generated test inputs on *faulty* versions of the method under test. To obtain such faulty versions, our current implementation employs MAJOR [28], which seeds artificial faults into the method under test, creating a set of k mutants of P: $\mu P_1, \mu P_2, \cdots \mu P_k$. Note that one could also add real faulty versions of P to the set of mutants.

3) Input Transformations. GENMORPH generates a set of input relations by relying on predefined templates that specify canonical input transformations [29]. These input relations are necessary for generating the follow-up inputs for Step 4.

4) Collection of Program Executions. GENMORPH instruments and executes each pair of source and follow-up inputs $\langle x_1, x_2 \rangle$ to capture at run-time the output values $P(x_1)$, $P(x_2)$ (correct executions) and $\mu P_i(x_1), \mu P_i(x_2)$ for each mutant $\forall i = 1, 2, \cdots k$ (incorrect executions). Collecting and caching such values is paramount to avoid the cost of re-executing all the tests when computing the FPs and FNs for each explored MR. Indeed, the evolutionary algorithm of GENMORPH might explore thousands of candidate

MRs. GENMORPH filters redundant correct and incorrect executions, as well as those incorrect executions that are equivalent to the correct ones obtained with the same input (i.e., it filters $\mu P_i(x_1)$ and $\mu P_i(x_2)$ if $\mu P_i(x_1) = P(x_1)$ and $\mu P_i(x_2) = P(x_2)$).

5) Generation of MRs. GENMORPH implements an evolutionary algorithm that, given an input relation and a set of correct/incorrect executions, explores candidate output relations to generate new ones, possibly with zero FP and the fewest FNs. The resulting input and output relations form a complete MR.

6) Filtering MRs. Step 5) outputs MRs with zero FPs with respect to the observed correct executions. Those MRs might fail for some other correct executions. To filter such MRs, GENMORPH has a filtering process that uses the test input generator RANDOOP [3] and the oracle assessor OASIS [20].

Until the time budget is reached, GENMORPH repeats steps 3) to 6) every time considering a different initial input relation.

4 RUNNING EXAMPLE

This section describes a running example of MR generation using GENMORPH, based on the *pow* function. We consider the implementation of the *pow* method from the Apache Commons Math library for Java [30] (see Listing 1): $pow(k, e)$. This method accepts two integers, k and e, and returns k^e.

One of the MRs generated by GENMORPH is[2]:

$$((k_f = k_s) \land (e_f = e_s - 1)) \Rightarrow (pow(k_f, e_f) = \frac{pow(k_s, e_s)}{k_s}) \quad (1)$$

Here, $x_1 = (k_s, e_s)$ denotes the *source* input, and $x_2 = (k_f, e_f)$ the *follow-up* input. In mathematical notation, this MR captures the following property of the *pow* function: $k^e = k^{e-1} \cdot k$.

Listing 1 shows a simplified version of the *pow* method. For this example, we will use one of the mutants generated by MAJOR, which we will refer to as pow_m. This mutant removes the `k2p *= k2p;` statement at line 12, resulting in some incorrect results. The code also shows the instrumentation added by GENMORPH to extract the method inputs and outputs (lines 2 and 15). Listing 2 shows a JUnit test case for *pow* generated by EVOSUITE. To collect *correct* test executions, GENMORPH executes the JUnit test cases generated by EVOSUITE with the instrumented *pow* method (without mutations). To collect *incorrect* test executions, GENMORPH executes the same JUnit test cases with the *pow* mutants generated by MAJOR (which are also instrumented in the same way). Listing 3 shows the data generated by the instrumentation after executing the test case in Listing 2

---

2. Slightly simplified for clarity and readability

```
1  public static int pow(final int k, long e) {
2      GenMorph.saveInputValues(k, e) // instrumentation
3      if (e < 0) {
4          throw new NotPositiveException();
5      }
6      int result = 1;
7      int k2p = k;
8      while (e != 0) {
9          if ((e & 0x1) != 0) {
10             result *= k2p;
11         }
12         k2p *= k2p; // Mutation: Remove line
13         e >>= 1;
14     }
15     GenMorph.saveOutputValues(result) //  instrumentation
16     return result;
17 }
```

Listing 1: Apache Commons Math - pow method (simplified)

```
1  @Test(timeout = 4000)
2  public void test100()  throws Throwable  {
3    int int0 = ArithmeticUtils.pow(−128, 2);
4  }
```

Listing 2: Test input for pow generated by EVOSUITE

on the instrumented *pow* method. This is the format for correct and incorrect test executions that GENMORPH uses.

Let us consider the input relation $((k_f = k_s) \wedge (e_f = e_s - 1))$. Given the source input $(k = -128, e = 2)$ (Listing 2), the follow-up input is $(k_f = -128, e_f = 1)$. Executing the original *pow* (Listing 1) with these source and follow-up inputs yields respectively the following (correct) outputs:

$$pow(-128, 2) = 16384 \quad pow(-128, 1) = -128$$

Executing the faulty version generated with MAJOR (pow_m) with the same inputs yields the following outputs:

$$pow_m(-128, 2) = -128 \quad pow_m(-128, 1) = -128$$

If we consider these correct and incorrect executions, GENMORPH infers that the generated MR shown in Eq.1 correctly classifies these executions. This MR identifies pow_m as faulty since the input relation is satisfied, but the output relation is violated. At the same time, the original *pow* satisfies the MR.

Obviously, there are too few test executions used in this example to capture the behavior of *pow*, so other (invalid) MRs could also be inferred from the considered executions, such as:

$$((k_f = k_s) \wedge (e_f = e_s - 1)) \Rightarrow (pow(k_f, e_f) \neq pow(k_s, e_s)) \quad (2)$$

This MR is incorrect because it yields FPs when $k = 0$ or $k = 1$. Consider the following input and output pair, generated with the same mutant: $pow_m(2, 8) = 2 \quad pow_m(2, 7) = 8$. The invalid MR (Equation 2) would pass, whereas our example MR (Equation 1) would correctly identify this faulty execution.

This example shows that intermediate MRs might have both FPs and FNs. That is why GENMORPH uses genetic

```
1      "systemId": "pow@original", "testId": "test100",
2      "variables": { "inputs": { "k": −128.0, "e": 2.0},
3                  "outputs": {"return": 16384.0} }
```

Listing 3: GenMorph (correct) execution by executing Listing 2

programming (GP) to improve candidate MRs until they have no FPs and fewer FNs.

5 INPUT TRANSFORMATIONS

GENMORPH starts the generation of MRs from the input relations. From a set of predefined transformation templates, GENMORPH selects a series of (metamorphic) input transformations compatible with the given method under test. For each selected transformation and source test input, GENMORPH generates a follow-up test input. Furthermore, it also generates a *canonical input relation* from each selected transformation. Such input relation is the most strict interpretation of the applied transformation: Given the values of the source test inputs, there is only a single possible value of the follow-up inputs that satisfies the canonical input relation.

These transformation templates are similar to what the MR literature refers to as metamorphic relation input patterns (MRIPs): An abstraction that characterizes the relations among the source and follow-up inputs of a set of MRs [24]. Our templates, however, define not only the input relations, but also the corresponding transformations from the source to the follow-up inputs.

Currently, GENMORPH implements the following **transformation templates**. We derived such templates by referring to the recent study of Duque et al. [29] that reports MRs commonly used in the literature (e.g., [31], [32]).

PermuteParameters. It permutes two of the input parameters of the method. It can be applied to any pair of method parameters, as long as they have the same type. For example, there is one possible instantiation for *pow*, in which the follow-up for $pow(k, e)$ is $pow(e, k)$. In this example, the canonical input relation would be: $(k_f = e_s) \wedge (e_f = k_s)$.

BooleanFlip. It flips the value of a Boolean parameter from *true* to *false* or vice versa. This transformation cannot be applied to *pow* because both of its parameters are numeric. Although the study of Duque et al. [29] does not mention this relation, we add it to support input relations of Boolean parameters.

NumericAddition(Number). It adds a constant number (positive or negative) to a single numeric parameter. For *pow*, we could apply `NumericAddition(-1)` to its second parameter to generate the follow-up input $pow(k, e - 1)$. This is the transformation template used by GENMORPH to generate the MR discussed in Section 4. The corresponding canonical input relation would be: $(k_f = k_s) \wedge (e_f = e_s - 1)$.

NumericMultiplication(Number). This transformation multiplies a single numeric parameter by a constant number (positive or negative). For *pow*, we can apply `NumericMultiplication(2)` to its first parameter and generate the follow-up input $pow(k \cdot 2, e)$. The corresponding input relation would be: $(k_f = k_s \cdot 2) \wedge (e_f = e_s)$.

SequenceRemove(Number). It removes a single element from a Sequence parameter. The numeric parameter of this template indicates the index to be removed, with negative indices being allowed for backwards indexing (like Python list indexing). If the index is out of bounds, this operation is a no-op. This transformation cannot be applied to *pow* because both of its parameters are numeric.

SequenceFlip. It inverts the order of the elements from a Sequence parameter. This transformation cannot be applied to *pow* because both of its parameters are numeric. Although the study of Duque et al. [29] does not mention this relation directly, it is conceptually similar to permuting parameters, and has been applied in various MRs throughout the literature.

Except for trivial functions (e.g., those that have a single Boolean parameter), there are many possible applicable input transformations to a given function (using the above templates). Indeed, the parameterized transformation templates (i.e., *NumericAddition* and *NumericMultiplication*) have a myriad of possible instantiations, as their parameter can be any number.

Because of this, it is necessary to sample a meaningful set of values for the parameterized transformation templates. To this aim, GENMORPH collects a pool of constants, consisting of the predefined values -1 and 1, plus all the constant values appearing in the method under test. GENMORPH extracts the constant values by instrumenting all the variable accesses and literal expressions of the method under test. Then it executes all the source inputs generated by EVOSUITE in Step 1, and identifies all the values logged by the instrumentation that remain the same in all executions. Such constants are likely to represent meaningful values for the method under test that might lead to semantically meaningful MRs. GEN-MORPH randomly samples the constant values to use with parameterized transformations (prioritizing constants that appeared more often).

6 GENERATION OF MRs

GENMORPH explores the space of possible MRs to find one that accurately classifies correct and incorrect test executions. Unfortunately, the space of possible MRs is enormous. As such, GENMORPH employs GP [33] to guide the search.

Similarly to GASSERT [25], GENMORPH implements a co-evolutionary algorithm that evolves two populations of MRs in parallel, with three competing objectives: (i) minimizing the FPs, (ii) minimizing the FNs, (iii) minimizing the size of the MR. Each population uses a different fitness function. The fitness function used in the first population (ϕ_{FP}) rewards MRs with fewer FPs, while the function of the second population (ϕ_{FN}) those with fewer FNs. Both populations consider the remaining two 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 $FP(MR)$ and $FN(MR)$ denote the false positive and false negative rate of an individual MR (with respect to the given correct and incorrect executions), and let $|MR|$ be the size of the MR (i.e., the number of syntactic elements in its predicate). The multi-objective fitness functions ϕ_{FP} and ϕ_{FN} [25] are defined using the concept of *dominance* ($\prec$) [34]:

Definition 4. **FP-fitness (ϕ_{FP})**. Given two metamorphic relations MR_1 and MR_2, MR_1 **dominates$_{FP}$** MR_2 ($MR_1 \prec_{FP} MR_2$) if any of the following conditions is satisfied:
  – $FP(MR_1) < FP(MR_2)$
  – $FP(MR_1) = FP(MR_2), FN(MR_1) < FN(MR_2)$
  – $FP(MR_1) = FP(MR_2), FN(MR_1) = FN(MR_2), |MR_1|<|MR_2|$

Algorithm 1:
GENMORPH CO-EVOLUTIONARY ALGORITHM

> **input** : R_i: canonical input relation
> correct $\mathcal{E}^+$ and incorrect $\mathcal{E}^-$ test executions
> **output**: $\mathbb{MR}$ a set of the best metamorphic relations

1 **function MR-GENERATION**
2    $Popul^{FP} \leftarrow$ GET-INITIAL-RANDOM-POPULATION(R_i)
3    $Popul^{FN} \leftarrow$ GET-INITIAL-RANDOM-POPULATION(R_i)
4    $gen \leftarrow 0$
5   **repeat**
6      $gen \leftarrow gen + 1$
7   **do in parallel**
8        $Popul^{FP} \leftarrow$ SELECT+REPRODUCE$(Popul^{FP}, \phi_{FP}, gen)$
9        $Popul^{FN} \leftarrow$ SELECT+REPRODUCE$(Popul^{FN}, \phi_{FN}, gen)$
10   **if** *gen % FREQ_MIGRATION = 0* **then**
11   add GET-BEST-MRS$(Popul^{FN}, \phi_{FN})$ to $Popul^{FP}$
12   add GET-BEST-MRS$(Popul^{FP}, \phi_{FP})$ to $Popul^{FN}$
13   **until** *time budget is expired*
14   **return** GET-BEST-MRS $(\{Popul^{FP} \cup Popul^{FN}\}, \phi_{FP})$

15 **function SELECT-+-REPRODUCE**
16    $Popul \leftarrow$ COMPUTE-FITNESS$(Popul, \phi, \mathcal{E}^+, \mathcal{E}^-)$
17    $Popul_{NEW} \leftarrow$ GET-BEST-MRS $(Popul, \phi)$ // elitism
18   **repeat**
19      $\langle MR_{p1}, MR_{p2} \rangle \leftarrow$ SELECT-PARENTS$(Popul, \phi)$
20      $\langle MR_{o1}, MR_{o2} \rangle \leftarrow$ CROSSOVER+MUTATION(MR_{p1}, MR_{p2})
       add $\langle MR_{o1}, MR_{o2} \rangle$ to $Popul_{NEW}$
21   **until** $Popul_{NEW}$ *is full*
22   **return** $Popul_{NEW}$

Definition 5. **FN-fitness (ϕ_{FN})**. Given two metamorphic relations MR_1 and MR_2, MR_1 **dominates$_{FN}$** MR_2 ($MR_1 \prec_{FN} MR_2$) if any of the following conditions is satisfied:
  – $FN(MR_1) < FN(MR_2)$
  – $FN(MR_1) = FN(MR_2), FP(MR_1) < FP(MR_2)$
  – $FN(MR_1) = FN(MR_2), FP(MR_1) = FP(MR_2), |MR_1|<|MR_2|$

In tie cases, $FP(MR_1)=FP(MR_2)$ and $FN(MR_1)=FN(MR_2)$, ϕ_{FP} and ϕ_{FN} favor smaller MRs, which are easier to understand.

Function **MR-GENERATION** of Algorithm 1 describes our co-evolutionary approach. First, GENMORPH initializes two distinct populations ($Popul^{FP}$ and $Popul^{FN}$) with MRs composed of the given input relation (R_i) and randomly generated output relations (R_o) (lines 2 and 3). Then, until the time budget is expired, GENMORPH evolves each population in parallel (Function SELECT+REPRODUCE). $Popul^{FP}$ uses ϕ_{FP} to select the individuals, while $Popul^{FN}$ uses ϕ_{FN}. Periodically, the two populations exchange their best individuals (lines 10 to 12).

6.1 Selection, Crossover, and Mutation

Function **SELECT+REPRODUCE** of Algorithm 1 describes how each generation of GENMORPH evolves a population of MRs ($Popul$ into $Popul_{NEW}$). First, it computes the fitness of each (new) individual in the population (Line 16). This amounts to counting the number of FPs, FNs, and computing the size of the MRs. Second, GENMORPH initializes the new population $Popul_{NEW}$ with the best individuals (elitism) to ensure that they will not be lost (Line 17). Then, Function SELECT+REPRODUCE applies Selection, Crossover, and Mutation. The **Selection** step selects two parent individuals $\langle MR_{p1}, MR_{p2} \rangle$ based on the given fitness function, ϕ_{FN} for $Popul^{FN}$ and ϕ_{FP} for $Popul^{FP}$ (Line 19). The **Crossover** step combines the genetic materials (portions of MRs in our case)

TABLE 1: Operators considered by GENMORPH

Operands	Output	Function
$\langle number, number \rangle$	$number$	$+, -, *, /$ (protected division)
$\langle number \rangle$	$number$	ABS
$\langle number, number \rangle$	$Boolean$	$==, \neq, <, >, \leq, \geq$
$\langle number \rangle$	$sequence$	toString
$\langle Boolean, Boolean \rangle$	$Boolean$	AND, OR, XOR, iff, implies
$\langle Boolean \rangle$	$Boolean$	NOT
$\langle sequence \rangle$	$number$	length, sum
$\langle sequence \rangle$	$boolean$	$==, \neq$
$\langle sequence \rangle$	$sequence$	flip
$\langle sequence, number \rangle$	$sequence$	remove, truncate

of the selected individuals and produces two new MRs (off-spring) $\langle MR_{o1}, MR_{o2} \rangle$ (Line 20). Finally, the **Mutation** step mutates (with a certain probability) the obtained offspring (Line 20) and adds them to the new population. It repeats these three steps until the new population is full.

6.1.1 Selection

GENMORPH implements two different selection criteria, and chooses between them with a given probability.

Tournament Selection [35] runs two "tournaments" among K random individuals. The winner of each tournament (the one with the highest fitness) is selected as a parent. We chose $K = 2$, as it mitigates the *local optima problem* [36].

Best-match Selection [25] is a selection criterion specific for test oracles. It selects the first parent randomly and selects the second parent with a higher probability if it maximizes the collective number of covered correct and incorrect executions.

6.1.2 Crossover and Mutation

GENMORPH represents output relations as rooted binary trees, where nodes can be operators, constant values, or variables. Table 1 shows the operators supported by GEN-MORPH. All numeric values are treated as real numbers (with floating point, fixed precision representation), and the numeric constants can have any real value in the range $[-100, 100]$. On the other hand, $true$ and $false$ are Boolean constants, and there are no sequence-type constants. As for the variables, the output relation can contain any input or output variable from the source or the follow-up inputs.

Crossover. GENMORPH relies on the classical tree-based crossover [33]. It selects a random crossover point in the output relations of each parent and creates two trees by swapping the subtrees rooted at each point.

Mutation. GENMORPH relies on three tree-based mutation operators (chosen randomly with a given prob.).

Node Mutation changes a single node in the tree [37]. Given a tree and one of its leaf nodes n, the new tree has n replaced with a new node of the same type (generated randomly).

Subtree Mutation replaces a subtree in the tree [37]. Given a tree and one of its inner nodes n, it returns a new tree obtained by substituting the subtree rooted at n with a randomly generated subtree of the same type.

Constant Value Mutation changes the value of a numeric constant node. 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 $\{-\Delta, \Delta\}$. Here, Δ should be small (we use 0.1 in our experiments) so that the constant values change in small increments.

6.2 Constraints

GENMORPH constrains the generated MRs. If an individual violates one or more of these constraints, it is dropped immediately, without evaluating its fitness. Currently, the output relations have a configurable complexity limit, which is the maximum number of nodes that its tree can have, and any individual which surpasses this limit will not be added to the population.

Furthermore, we also implemented a "soft" constraint, which is required for an individual to be considered for the elite set or as the best individual, but not for being added to the population. This constraint is that the output relation must contain both the source and the follow-up variables for at least one of the method outputs. This filters out expressions that are not truly MRs, as MRs are supposed to check the outputs from both the source and the follow-up test cases. Although individuals that violate this constraint are not MRs, they are allowed into the population because they may contain useful genetic material.

Finally, we also added uniqueness constraints for elitism. Specifically, an individual must satisfy two requirements to be included in the elite population, on top of its fitness. First, there must not be another individual with an identical output relation already in the population. Second, there must not be another individual with an identical set of FNs already in the population. These constraints prevent semantically equivalent individuals from taking multiple spots in the elite population. Notably, the second constraint is needed because simple mutations can easily bypass the first constraint and generate semantically equivalent individuals. For instance: $ASSERTION$ may become $(ASSERTION \wedge true)$.

7 MR FILTERING

GENMORPH performs a final filtering process to avoid reporting *invalid* MRs to the user. The validity of an MR is determined by the lack of FPs after a 2-step filtering process.

The first step validates the MRs with new automatically-generated test inputs unseen by the evolutionary algorithm. We use the random test generator RANDOOP [3] to generate the new source inputs. Then, GENMORPH can simply generate the corresponding follow-up inputs using the template-based input transformation corresponding with the input relation of each MR.

For each MR under analysis, GENMORPH creates a JUnit test suite with a test case for each source and follow-up input pair generated with RANDOOP and the input transformation for the MR. Listing 4 shows a JUnit test case generated for the example MR shown in Eq. 1. Each failing test would be a FP for the analyzed MR.

The second step relies on OASIs [20], [21], [22], an oracle assessor designed to detect FPs and FNs in program assertions. To detect FPs, it negates the assertion and creates a new branch at the assertion point with the negated condition. It then employs search-based test generation to find test cases that cover that branch, which would represent FPs of the assertion.

To adapt OASIs' functionality to MRs, we changed its original implementation. Given an MR, OASIs creates a new method that, similarly to Listing 4, has an if statement with the input relation as its condition and an assert statement

```
1  @Test
2  public void test0followup() {
3      int k_s = -128, e_s = 2; // source input
4      int k_f = -128, e_f = 1; // follow-up input
5      int o_s = pow(k_s, e_s); // run source test input
6      int o_f = pow(k_f, e_f); // run follow-up test input
7      if ((k_f == k_s) && (e_f == e_s - 1)) {//Ri is true
8          assertTrue(o_f == (o_s / k_s)); // check Ro
9      }
10 }
```

Listing 4: GENMORPH-generated executable MR of Eq. (1)

with the output relation as its body. However, unlike Listing 4, the generated method is not input specific, but takes the source and follow-up inputs as parameters. This new branch becomes the target of the search-based test generation of OASIs. This enables the detection of FPs, as the inputs that make the assert statement fail will need to satisfy the input relation first.

Note that the filtering step ignores the FNs of MRs. As discussed in Section 2, FNs are inevitable in MRs, while ensuring absence of FPs is crucial to avoid false alarms.

8 EVALUATION

Our evaluation aims to answer three research questions (RQs).

RQ1: Effectiveness Is GENMORPH effective at synthesizing valid and useful metamorphic relations automatically?

RQ2: Test Case Oracle Enhancement How much does GENMORPH increase the fault detection capability of automatically generated test inputs and test case oracles?

RQ3: AutoMR Comparison How do the results from GEN-MORPH compare with those obtained by AUTOMR [18]?

RQ4: Filtering Is the filtering process of GENMORPH effective at detecting invalid metamorphic relations?

RQ1 focuses on the main objective of GENMORPH, which is the generation of oracles that do not trigger false alarms on the original code (*valid* oracles), while being effective at exposing faults injected by mutation analysis (*useful* oracles). To be effective, metamorphic oracles should also pass the OASIs filter, which excludes metamorphic oracles for which evidence of false alarms can be automatically generated. RQ2 compares the fault detection capability of GENMORPH with that of test generators that produce test cases with assertions on the observed execution output for specific test inputs (henceforth referred to as *test case oracles*). Similarly to the regression oracles generated by GENMORPH, which capture universal properties, these test case oracles can be useful to expose faults in a regression testing context. RQ3 compares GENMORPH with AUTOMR [18], a state-of-the-art MR generation tool that can generate polynomial MRs for programs with numeric inputs and outputs. RQ4 focuses on the usefulness of the the proposed filtering process, which uses automated test generation to find evidence of false positives and to filter out the corresponding metamorphic oracles.

8.1 Experimental Setup

Experimental Subjects. We evaluated GENMORPH on three different open-source Java libraries: (1) Apache Commons

Math 3 [38], (2) Apache Commons Lang 3 [39], and (3) Google Guava 31 [40]. We randomly selected the methods from the ones in the libraries that fulfill the following criteria: (1) it contains at least 8 lines of code. The methods we selected contain between 8 and 93 LOC; (2) it is a static method (since GENMORPH does not currently consider the `this` object state); (3) all the method parameters are currently supported by GENMORPH. The supported types include Java primitives (e.g., `boolean`, `int`) and their corresponding wrappers (e.g., `java.lang.Boolean`, `java.lang.Integer`), or types which we consider sequences in our implementation (arrays, `java.lang.List` implementations, or `java.lang.String`); (4) it does not use I/O or non-deterministic operations (e.g., reading/writing files, using random numbers). The 23 selected methods span 13 different classes from the three different libraries. The first four columns of Table 2 show their library, name, signature, LOC, # of mutants.

Setup. We now describe the evaluation setup for GEN-MORPH.

Step 1) Source Input Generation. For each subject, we ran EVOSUITE [2] (v. 1.1.0) to generate the source test inputs. We configured EVOSUITE with the branch coverage criterion, minimization enabled, and a time budget of 5 minutes. We performed ten runs with different random seeds and aggregated all the test cases to obtain a diverse and large set of test inputs.

Step 2) Mutants Generation. For each subject, we ran MAJOR [28] (v. 2.0.0) enabling all types of supported mutants. We automatically filtered the mutants by mutated lines of code, keeping only those that directly affect the method under test.

Step 3) Input Transformation. In order to keep the MR generation at a reasonable cost, we configure GENMORPH to select four instantiations of our templates for each run.

Step 4) Collection of Program Executions. Since the cost of evaluating the fitness of an MR grows linearly with the number of correct and incorrect executions, we enforced a limit of 9,000 unique correct and 9,000 unique incorrect executions. If more are obtained, the executions are randomly sampled.

Step 5) Generation of MRs. Table 3 shows the configuration of Algorithm 1.

For each subject method and strategy, we used a time budget of 30 minutes and we generated four different MRs. We repeated each run 12 times due to the stochasticity of the approaches.

Step 6) Filtering MRs. For each subject, we ran RAN-DOOP [3] (v. 4.3.0) with 12 different random seeds. In order to limit the run-time of the experiments to a reasonable cost, we sampled 100 RANDOOP test cases for each run. We ran OASIs [20] multiple times (due to its stochastic nature) with an overall budget of 10 minutes, giving each run a budget of 150 seconds. We stop OASIs when we find the first FP.

We ran the experiments on four VMs with a 6-core AMD Zen 3.2 GHz CPU and 12 GB of RAM.

8.2 RQ1: Effectiveness

RQ1 evaluates the effectiveness of GENMORPH at synthesizing effective MRs using Mutation Score (MS) (i.e., the ratio of

TABLE 2: Evaluation results (average) (MS = Mutation Score, PZ = Ratio of MRs without FPs, PZO = Ratio of MRs without FPs with OASIs, ΔMS = GENMORPH's MS over mutants missed by RANDOOP [3] (ΔMS_R), EVOSUITE [2] (ΔMS_E) or AUTOMR [18] (ΔMS_A)).

library	method	signature	# mutants	Randoop MS	Evosuite MS	AutoMR MS	GENMORPH MS	ΔMS_R	ΔMS_E	ΔMS_A	PZ	PZO
Math	nextPrime	`int(int)`	20	0.80	0.90	0.28	0.78	0.75	0.46	0.72	0.70	0.97
Math	isPrime	`bool(int)`	25	0.52	0.92	0.28	0.29	0.67	0.06	0.17	0.32	1.00
Math	gcd	`int(int,int)`	25	0.50	0.83	0.00	0.63	0.59	0.05	0.63	0.90	1.00
Math	pow	`int(int,int)`	10	0.70	1.00	0.00	0.69	0.00	-	0.69	0.78	1.00
Math	stirling	`long(int,int)`	46	0.28	0.56	0.00	0.31	0.24	0.20	0.31	0.69	0.79
Math	acos	`double(double)`	76	0.93	0.50	0.00	0.09	0.02	0.08	0.09	0.05	0.71
Math	log10	`double(double)`	15	1.00	0.60	0.00	0.06	-	0.00	0.06	0.03	0.32
Math	sin	`double(double)`	26	0.71	0.73	0.41	0.60	0.00	0.00	0.36	0.70	0.97
Math	sinh	`double(double)`	123	0.89	0.41	0.23	0.21	0.00	0.10	0.20	0.25	0.55
Math	tan	`double(double)`	37	0.76	0.70	0.32	0.38	0.00	0.01	0.25	0.56	0.79
Lang	abbreviate	`string(string,string,int,int)`	39	0.85	0.30	-	0.41	0.01	0.13	-	0.53	0.96
Lang	capitalize	`string(string)`	10	0.90	0.41	-	0.28	0.00	0.00	-	0.13	1.00
Lang	center	`string(string,int,string)`	12	0.83	0.25	-	0.53	0.00	0.28	-	0.64	0.94
Lang	difference	`string(string,string)`	6	0.67	0.36	-	0.24	0.00	0.50	-	0.19	0.90
Lang	isSorted	`bool(int[])`	11	0.90	0.37	-	0.18	0.25	0.25	-	0.11	0.80
Guava	indexOf	`int(bool[],bool[])`	12	1.00	1.00	-	0.14	-	-	-	0.04	0.40
Guava	join	`string(string,bool[])`	5	0.80	0.80	-	0.13	0.00	0.00	-	0.06	1.00
Guava	meanOf	`double(int[])`	12	0.96	1.00	-	0.83	0.00	-	-	0.88	1.00
Guava	min	`int(int[])`	9	0.89	0.89	-	0.55	0.00	0.00	-	0.47	0.74
Guava	padStart	`string(string,int,char)`	7	0.86	0.83	-	0.64	0.00	0.16	-	0.35	0.85
Guava	repeat	`string(string,int)`	18	0.94	0.81	-	0.86	0.00	0.69	-	0.67	0.97
Guava	sort	`void(byte[],int,int)`	8	0.95	0.88	-	0.58	0.00	0.00	-	0.51	0.83
Guava	truncate	`string(string,int,string)`	10	1.00	0.78	-	0.28	-	0.18	-	0.23	0.80

killed mutants). For each MR returned by Step 5, we ran the 2-step filtering process (RANDOOP, and OASIS, see Section 7). If the filtering process did not find FPs, we passed each of the 12 test suites derived from the RANDOOP test inputs and the generated MR to PIT [41] (v. 1.7.4) for computing the MS. This results in 12 different MS measures for each MR. If the 2-step filtering process found FPs, we considered the MS to be zero. Note that during MR generation, GENMORPH relies on completely separate tools (i.e., EVOSUITE and MAJOR) to obtain the incorrect executions. In the empirical study, using different test inputs and mutants to compute the MS is important to properly separate the training phase from the evaluation phase.

Table 2 shows the average **MS** (Mutation Score) obtained with the MRs generated by GENMORPH for each subject method. Such average MS ranges between 6% and 86%. Furthermore, GENMORPH achieves an average MS higher than 20% in 18 out of the 23 methods and higher than 50% in 10 out of the 23 methods. Since the MRs generated by GEN-MORPH are reusable properties of the methods that can be integrated into any existing test suite, or even implemented as runtime checks, we consider an MS of 20% to be effective enough, and 50% to be very effective.

Table 2 shows that the **PZ** (ratio of MRs without FPs) ranges between 3% and 90%. For 15 out of the 23 methods, PZ was no less than 25%, which indicates that at least one valid MR is generated on average in a single run of GENMORPH with our configuration. Generally, all the methods where GENMORPH achieves a low MS are explained by a similarly

TABLE 3: Configuration Parameters of Algorithm 1

Parameter Description	Value	Parameter Description	Value
time budget (minutes)	30	prob. of crossover	90%
max correct executions	9,000	prob. of mutation	30%
max incorrect executions	9,000	frequency of migration (every X gen)	10
bound on the size of R_o	16	number of individuals for elitism	10
size of each of the populations	1,000	number of individuals to migrate	160

low PZ, which might indicate that the dataset of correct test executions used for generating MRs for those methods was not comprehensive enough.

> **RQ1 – In summary:** GENMORPH generated valid and effective MRs for the majority of the subject methods.

8.3 RQ2: Test Case Oracle Enhancement

RQ2 evaluates the degree to which the generated MRs can improve the fault detection capability of test inputs and test case oracles generated by RANDOOP and EVOSUITE. To achieve this, we ran RANDOOP and EVOSUITE 12 times with assertions enabled, using the same configuration we used to generate test suites for the MRs with RANDOOP. Columns **RANDOOP MS** and **EVOSUITE MS** in Table 2 show the mean mutation score (MS) obtained by the RANDOOP and EVOSUITE test suites.

Automated test generators like RANDOOP produce test case oracles that capture the observed behavior of the generated test inputs to expose regression faults as the software evolves. However, test case oracles only capture the expected behavior for a *specific* input. Differently, MRs represent universal invariants that can be used to enhance any test suite.

We can notice that the MS values in Table 2 are often comparable, with a tendency of test case assertions to achieve higher MS values. However, the MS values of RANDOOP and EVOSUITE are obtained by substantially different types of oracles w.r.t. GENMORPH, the former being test case assertions that predicate on single executions, and the latter metamorphic relations that hold for all program executions. To assess the degree of complementarity of these different types of oracles, we considered the mutants missed by RANDOOP or EVOSUITE and checked whether they are killed by GENMORPH.

Specifically, we automatically enhanced the test suites generated by RANDOOP and EVOSUITE by adding all the MRs generated by GENMORPH. Each MR takes a source input produced by the test generation tools, generates the corresponding follow-up test input, and checks if the output relation is satisfied. In Table 2, columns **GENMORPH** $\Delta\mathbf{MS}_R$ and **GENMORPH** $\Delta\mathbf{MS}_E$ show the percentage of the survived mutants (i.e., those not killed by RANDOOP or EVOSUITE test inputs and oracles) that are killed by the enhanced test suite. We can notice that GENMORPH's MRs are highly complementary to the test case assertions generated by RANDOOP and EVOSUITE, as respectively in 7 out of 23 and in 14 out of 23 cases the inclusion of GENMORPH's MRs produce a mutation score improvement (ΔMS > 0).

Let us consider a few examples. For `nextPrime`, RANDOOP test suites alone kill 80% of the mutants. When this test suite is enhanced with the MRs generated by GENMORPH, the mutation score is 95%, which means that the enhanced test suites kill 75% of the remaining mutants, i.e., ΔMS$_R$ = 75% = ($\frac{95\%-80\%}{100\%-80\%}$).

The results show a great improvement of the RANDOOP MS for `nextPrime`, `isPrime` and `gcd` (ΔMS$_R$ between 59% and 75%), as well as a significant improvement for `stirlings` and `isSorted` (ΔMS$_R$ of 24% and 25%) and a small improvement for `acos` and `abbreviate` (ΔMS$_R$ of 2% and 1%). Although the MRs do not kill any additional mutants for the remaining methods, RANDOOP's MS is already 100% for `log10`, `indexOf` and `truncate`, so improvement is not possible for those methods.

As for EVOSUITE, the results show a great improvement for the `nextPrime`, `difference` and `repeat` methods (ΔMS$_E$ of 46%, 50% and 69%, respectively), as well as a significant improvement for seven other methods (ΔMS$_E$ between 10% and 28%) and a small improvement for four other methods (ΔMS$_E$ between 1% and 8%). EVOSUITE's MS is 100% for `pow`, `indexOf` and `meanOf`, making it impossible any improvement in those methods.

> **RQ2 – In summary:** GENMORPH generated MRs that enhance the fault detection capability of both RANDOOP and EVOSUITE-generated regression test inputs. This shows the complementarity between GENMORPH's MRs and RANDOOP and EVOSUITE's test case assertions.

8.4 RQ3: AutoMR Comparison

RQ3 compares GENMORPH with the AUTOMR [18] numeric MR generation tool.

AUTOMR [18] is based on MRI [19], a search-based approach to automatically generate polynomial MRs. MRI supports MRs in which the input relation is a linear function, and the output relation is either linear or quadratic. MRI calculates the optimal coefficients of the linear and quadratic equations using *particle swarm optimization* [42].

MRI uses 1,000 randomly-generated test inputs to filter the MRs that fail for a non-negligible percentage of those inputs. AUTOMR [18] addresses several limitations of MRI: MRI does not support MRs with (i) inequality, (ii) more than one input, and (iii) relations of higher degrees than quadratic.

Like MRI, AUTOMR uses particle swarm optimization to search for the MR parameters. Similar to GENMORPH,

AUTOMR generates MRs as pairs of input and output relations, although the latter defines both as arbitrary-degree polynomial relations, and parameterizes both within the same search process. Since AUTOMR is an improvement of the approach used by MRI, we compare the effectiveness of GENMORPH against AUTOMR only.

We employ the code implemented by the AUTOMR authors[3] in our evaluation. Since this tool is implemented in Python, we implemented a remote function call protocol[4] in order to allow this tool to call the experimental subjects, which are Java methods. Since AUTOMR calls the subject methods during the search (whereas GENMORPH calls them before) and the remote function call protocol introduces some additional overhead, we did not implement a fixed time budget as with GENMORPH, but instead employed the same configuration used by the AUTOMR authors: 3 PSO runs with maximum iterations of 350 for each experiment.

AUTOMR has several additional parameters to select for the generated MR, namely: (1) Number of inputs involved, (2) mode of input relation (equality, greater-than, less-than), (3) mode of output relation, (4) degree of input relation (linear, quadratic, etc.), and (5) degree of output relation. We employ the 13 different parameterizations that were defined in the settings from the AUTOMR codebase, and also left other configurations unchanged. Same as for GENMORPH, we repeated each run 12 times with different random seeds due to the stochasticity of AUTOMR.

As for the experimental subjects, we ran AUTOMR on the 10 methods from the Apache Commons Math 3 library. The reason why we evaluated only these methods is that they are purely numeric and contain no variable-length parameters, as it is unclear how AUTOMR would be applied to methods that do not fulfill this criteria. In order to convert these subject methods to purely numeric ones, Boolean values (output of the `isPrime` method) were converted to 0 (false) and 1 (true), and Java Exceptions were converted to 0 outputs.

To evaluate the MRs generated by AUTOMR, we performed the same PIT runs that we used for GENMORPH, with the same seed inputs. In order to avoid potential implementation issues with the MR evaluation, the AUTOMR test suites employed a remote protocol to compute the verdicts in Python, where code derived from AUTOMR was used to provide a pass or fail result. Our verdict function employs the same formulas used by AUTOMR for evaluating the output relations:

$$\mathbb{R}_{output} : |\mathbf{Bv}| < 0.05 \qquad \text{(AutoMR equality OR)}$$
$$\mathbb{R}_{output} : \mathbf{Bv} > 0 \qquad \text{(AutoMR greater-than OR)}$$
$$\mathbb{R}_{output} : \mathbf{Bv} < 0 \qquad \text{(AutoMR less-than OR)}$$

Column **AUTOMR MS** of Table 2 shows that GENMORPH achieves a higher mutation score in 9 out of the 10 subject methods for which AUTOMR was run. Furthermore, the difference in the method where GENMORPH was outperformed (`sinh`) is very small, with MSs of 21% for GENMORPH and 23% for AUTOMR. Note that AUTOMR produced no output for 5 out of the 10 subject methods, as all the generated MRs

---

3. https://github.com/bolzzzz/AutoMR
4. https://github.com/jonayerdi/JavaPythonBridge

were filtered out during the final redundant MR removal phase, hence the 0.00 values in the AUTOMR MS column.

Column **GENMORPH** Δ**MS**$_A$ shows the percentage of the survived mutants (i.e., those not killed by AUTOMR MRs) that are killed by GENMORPH MRs. These results show that GENMORPH MRs could identify several additional mutants that could not be detected with AUTOMR in all the subject methods, with ΔMS$_A$ values ranging between 17% and 72% for the subject methods where both tools produced effective MRs.

> **RQ3 – In summary:** GENMORPH generated MRs that achieved a higher mutation score than those generated by AUTOMR for 9 out of 10 subject methods. The MRs generated by GENMORPH kill more mutants than the MRs generated by AUTOMR in all subject methods.

8.5 RQ4: Filtering

RQ4 evaluates the effectiveness of the filtering process in detecting invalid MRs.

Table 2 shows that the **PZ** values are not greater than 86% for any of the methods, so at least 14% of the generated MRs are always filtered. In fact, more than 50% of the MRs are filtered in 13 out of 23 methods. On the one hand, we observe that the PZ is particularly low in methods involving boolean inputs or outputs. This might indicate that methods with boolean inputs or outputs can hold properties that seem valid for a large variety of inputs, but do not actually generalize for all possible inputs. Furthermore, since booleans only have two possible values, GENMORPH may generate more specific output relations, which are more prone to FPs. On the other hand, we also noticed that the filtering process eliminated most MRs for the `acos` and `log10` methods, which involve floating-point arithmetic. A possible reason could be that floating-point arithmetic has special values such as NaN or Infinity, which may invalidate some MRs which would be valid for the cases where all inputs and outputs are finite values.

To provide more insights on the contribution of OASIs for identifying invalid MRs, Table 2 also shows the **PZO** metric, the ratio of MRs for which OASIs does not find FPs. Note that only the MRs which yield no FPs with all the Randoop-generated test inputs are passed to OASIs, hence PZO is a ratio calculated over the MRs which already passed the Randoop filter. The results shows that OASIs can identify additional FPs not found by Randoop in 17 of the 23 subject methods. For `log10`, `sinh` and `indexOf`, OASIs identified FPs in more than 40% of the MRs that passed the Randoop filter, indicating that Randoop regression test suites alone may be insufficient to discard invalid MRs in some cases.

> **RQ4 – In summary:** GENMORPH's filtering process is effective in filtering MRs that do not generalize well on unseen inputs, and OASIs can successfully identify invalid MRs that are not identified with Randoop alone.

8.6 Threats to Validity

External validity. An external threat in our evaluation relates to the generalization of the results. We mitigated this threat by selecting 23 diverse functions from three different codebases. Moreover, we evaluated GENMORPH with unseen test inputs and seeded faults generated with different tools from the ones used to generate the MRs.

Internal validity. Internal validity threats arise from errors in the measurements or the implementation of GENMORPH. We mitigate this threat by manually inspecting the output and internal behavior of GENMORPH for a few sample runs.

Conclusion validity. The approach and evaluation methods are inherently stochastic. To account for this, we ran each technique 12 times and evaluated each generated MR with 12 different test suites, resulting in $12 \times 12 = 144$ samples.

9 RELATED WORK

Automatic test generation. Test generators are often meant for regression testing scenarios, and therefore capture the implemented behavior of the software under test in order to expose new faults as the software evolves [2], [3], [11], [43]. Similarly, GENMORPH also captures the implemented behavior of the system. In fact, GENMORPH's FN-based guidance is effectively very similar to the mutation-guided assertion generation used by Evosuite [2], since the states used to compute FNs in GENMORPH are obtained with mutation testing. Unlike test generators, however, GENMORPH produces universal properties of the system (MRs) that can be tested for any input, instead of input-specific assertions.

Metamorphic Testing. Most metamorphic testing approaches assume the availability of MRs [10]. For instance, much work has been done to predict whether an MR picked from a predefined list is suitable for a given program (e.g., [29], [44], [45], [46], [47], [48], [49]). Differently, GENMORPH aims to automatically generate new MRs for a given program, which remains a less studied problem [9], [10], [50]. We now discuss the most related work to the generation of new MRs.

ML-based approaches. Researchers investigated the use of machine-learning to predict whether specific types of MRs [51] hold for a given method [46], [52], [53]. These approaches only predict whether pre-defined types of MRs hold for a method. Developers are expected to derive an executable MR using this information [52]. Conversely, GENMORPH automatically generates executable MRs.

NLP-based approaches. MEMO [54] automatically derives equivalence MRs from the JavaDoc. The quality of derived MRs, by design, strictly depend on the completeness and correctness of the available documentation. Differently, GEN-MORPH does not rely on documentation and it is not limited to MRs expressed as equivalences.

Search-based approaches. MRI [19] and AUTOMR [18] are search-based approaches which employ particle-swarm optimization [42] to parameterize polynomial input and output relations.

Same as GENMORPH, MRI and AUTOMR use search-based algorithms to generate MRs for numerical programs. However, the fitness functions of MRI and AUTOMR take into account only the number of FPs. GENMORPH's fitness functions also consider the number of FNs, which is crucial to obtain MRs that are effective at exposing software faults. Moreover, GENMORPH is not limited to polynomial MRs, but can also operate on logical predicates and ordered sequences. We have conducted a comparison between AUTOMR and

GENMORPH with numerical subject methods only, since non-numeric functions are not supported by AUTOMR.

GASSERT [25] and EVOSPEX [26] generate oracles with evolutionary algorithms driven by both FPs and FNs. However, they target program assertions and invariants, respectively, and they cannot be easily adapted to target MRs. Moreover, their representation of FPs and FNs is ill-suited for MRs as discussed in Section 2. Ayerdi et al. proposed GASSERTMRS [55], which adapts GASSERT [25], [56] for generating MRs for CPS, considering, in particular, an industrial system of elevators as case study. However, GASSERTMRS supports domain-specific, system-level MRs that relate configuration variables for system level test scenarios (e.g., the number of elevators or passengers in each floor) with system level quality metrics (e.g., average waiting time). On the contrary, GENMORPH operates at the unit level and generates MRs that predicate on local variables of the method under test.

10   CONCLUSIONS AND FUTURE WORK

This paper presented GENMORPH, the first generator of metamorphic relations capable of minimizing at the same time the false positive rate (associated with false alarms) and the false negative rate (associated with missed faults). It includes a filtering phase that further eliminates MR oracles prone to residual false positives.

Our empirical results show that our evolutionary approach generates useful and non-trivial MRs, which can expose faults simulated by mutants in the majority of the analyzed subjects. The metamorphic oracle produced by GENMORPH has been shown to be highly complementary to the test case assertions automatically generated by test generators like RANDOOP and EVOSUITE. In fact, when the MRs generated by GENMORPH are added to RANDOOP and EVOSUITE's test cases, they increase the fault detection capability of the latter tools by a substantial amount. We have also evaluated the usefulness of the filtering phase, showing in particular that the oracle validator OASIs gives a fundamental contribution to the elimination of candidate oracles that trigger false alarms.

Future works aim at increasing the effectiveness and applicability of GENMORPH. We discuss the most promising ones.

Extending GENMORPH to handle complex types. While the general design of GENMORPH allows for arbitrarily complex expressions and type systems, our current implementation for Java methods only supports Boolean, numeric and ordered-sequence types for variables and operations. More research is needed to support complex types (e.g. user-defined classes) and their operations in the generated MRs.

Improving MRs readability. We found that several MRs generated by GENMORPH could be simplified to increase their readability. For example, by removing tautologies, reordering the variables, or replacing sub-expressions with equivalent but shorter ones. An important future work is to investigate post-processing analyses to simplify the generated MRs.

REFERENCES

[1]   E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[2]   G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[3]   C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*.   IEEE, 2007, pp. 75–84.

[4]   S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *International Symposium on Search Based Software Engineering*.   Springer, 2020, pp. 9–24.

[5]   P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[6]   A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[7]   M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

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

[9]   T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys*, vol. 51, no. 1, pp. 4:1–4:27, Jan. 2018.

[10]   S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, Sept 2016.

[11]   S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15.   IEEE Press, 2015, p. 201–211. [Online]. Available: https://doi.org/10.1109/ASE.2015.86

[12]   T. Y. Chen and T. Tse, "New visions on metamorphic testing after a quarter of a century of inception," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1487–1490.

[13]   J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer *et al.*, "Testing web enabled simulation at scale using metamorphic testing," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021.

[14]   Google, "graphicsfuzz," https://github.com/google/graphicsfuzz, 2022.

[15]   D. C. Jarman, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing for adobe data analytics software," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, May 2017, pp. 21–27.

[16] M. Lindvall, D. Ganesan, R. Ardal, and R. Wiegand, "Metamorphic model-based testing applied on nasa dat – an experience report," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2, May 2015, pp. 129–138.

[17] T. Y. Chen, "Metamorphic testing: A simple approach to alleviate the oracle problem," in *Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE), 2010*, June 2010, pp. 1–2.

[18] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato, "Automatic discovery and cleansing of numerical metamorphic relations," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 235–245.

[19] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 701–712. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642994

[20] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 2016, pp. 247–258.

[21] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, 2019.

[22] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Oasis: oracle assessment and improvement tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 368–371.

[23] J. Ayerdi, V. Terragni, G. Jahangirova, A. Arrieta, and P. Tonella, "Replication package," https://doi.org/10.5281/zenodo.10067096, 2023, last access: Nov 2023.

[24] 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. [Online]. Available: https://doi.org/10.1109/TSE.2018.2876433

[25] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1178–1189.

[26] F. Molina, P. Ponzio, N. Aguirre, and M. Frias, "Evospex: An evolutionary algorithm for learning postconditions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1223–1235.

[27] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[28] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 433–436.

[29] A. Duque-Torres, D. Pfahl, R. Ramler, and C. Klammer, "A replication study on predicting metamorphic relations at unit testing level," in *2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 698–708.

[30] Apache, "Apache Commons Math - pow method," https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/util/ArithmeticUtils.html\#pow(int,\%20int), 2023.

[31] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 1–10.

[32] U. Kanewala, "Techniques for automatic detection of metamorphic relations," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 237–238.

[33] J. R. Koza and J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT press, 1992, vol. 1.

[34] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[35] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Systems*, vol. 9, no. 3, pp. 193–212, 1995.

[36] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, vol. 4, no. 2, pp. 65–85, 1994.

[37] M. F. Brameier and W. Banzhaf, "A comparison with tree-based genetic programming," *Linear Genetic Programming*, pp. 173–192, 2007.

[38] Apache, "Apache Commons Math," https://commons.apache.org/proper/commons-math/, 2022.

[39] ——, "Apache Commons Lang," https://commons.apache.org/proper/commons-lang/, 2022.

[40] Google, "Google Guava," https://guava.dev/, 2022.

[41] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[42] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[43] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *2010 Third international conference on software testing, verification and validation*. IEEE, 2010, pp. 137–146.

[44] B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*. IEEE, 2018, pp. 14–17.

[45] K. Rahman and U. Kanewala, "Predicting metamorphic relations for matrix calculation programs," in *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*. IEEE, 2018, pp. 10–13.

[46] P. Zhang, X. Zhou, P. Pelliccione, and H. Leung, "Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network," *IEEE access*, vol. 5, pp. 21 791–21 805, 2017.

[47] K. Rahman, I. Kahanda, and U. Kanewala, "Mrpredt: Using text mining for metamorphic relation prediction," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 420–424.

[48] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 118–128.

[49] A. Nair, K. Meinke, and S. Eldh, "Leveraging mutants for automatic prediction of metamorphic relations using machine learning," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 1–6.

[50] Z. Hui and S. Huang, "Achievements and challenges of metamorphic testing," in *Fourth World Congress on Software Engineering (WCSE), 2013*, Dec 2013, pp. 73–77.

[51] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[52] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013*, Nov 2013, pp. 1–10.

[53] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Software Testing, Verification and Reliability*, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1594

[54] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "Memo: Automatically identifying metamorphic relations in javadoc comments for test automation," *J. Syst. Softw.*, vol. 181, p. 111041, 2021.

[55] J. Ayerdi, V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel, "Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1264–1274. [Online]. Available: https://doi.org/10.1145/3468264.3473920

[56] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Gassert: A fully automated tool to improve assertion oracles," in *Proceedings of the 43nd IEEE/ACM International Conference on Software Engineering, Demonstration Track ICSE-DEMO 2021, Virtual Event, Spain, 24 May – 1 June, 2021 (to appear)*, 2021.