Bug or not Bug? Analysing the Reasons Behind Metamorphic Relation Violations

Alejandra Duque-Torres[†], Dietmar Pfahl[†], Claus Klammer[‡] and Stefan Fischer[‡]

[†]Institute of Computer Science, University of Tartu, Tartu, Estonia

E-mail: {duquet, dietmar.pfahl}@ut.ee

[‡]Software Competence Center Hagenberg (SCCH) GmbH, Hagenberg, Austria

E-mail: {claus.klammer, stefan.fischer}@scch.at

Abstract—Metamorphic Testing (MT) is a testing technique that can effectively alleviate the oracle problem. MT uses Metamorphic Relations (MRs) to determine if a test case passes or fails. MRs specify how the outputs should vary in response to specific input changes when executing the System Under Test (SUT). If a particular MR is violated for at least one test input (and its change), there is a high probability that the SUT has a fault. On the other hand, if a particular MR is not violated, it does not guarantee that the SUT is fault free. However, deciding if the MR is being violated due to a bug or because the MR does not hold/fit for particular conditions generated by specific inputs remains a manual task and unexplored. In this paper, we develop a method for refining MRs to offer hints as to whether a violation results from a bug or arises from the MR not being matched to certain test data under specific circumstances. In our initial proof-of-concept, we derive the relevant information from rules using the Association Rule Mining (ARM) technique. In our initial proof-of-concept, we validate our method on a toy example and discuss the lessons learned from our experiments. Our proof-of-concept demonstrates that our method is applicable and that we can provide suggestions that help strengthen the test suite for regression testing purposes.

Index Terms—Metamorphic testing, metamorphic relation, association rule mining, passive testing.

I. INTRODUCTION

Software testing is a crucial stage in the software development life cycle as it ensures the software's proper operation and quality. One of the most significant challenges in software testing is the test oracle problem. A test oracle determines the System Under Test's (SUT) output for a given input. The test oracle problem occurs when the SUT lacks an oracle or when creating one to verify the computed outputs is practically impossible [1]. *Metamorphic Testing* (MT) is a software testing approach proposed by Chen *et al.* [2] to alleviate the test oracle problem.

In contrast to traditional testing techniques, MT examines the relations between input-output pairs of consecutive SUT executions rather than the individual outputs. The relations between SUT inputs and outputs in MT are known as *Metamorphic Relations* (MRs). MRs specify how the outputs should vary in response to specific input changes [3]. When an MR is violated for at least one test input and its change, there is a strong likelihood that the SUT has a fault. However, the absence of violation does not ensure that the SUT is fault-free. As a result, the suitability of the MRs employed significantly impacts the effectiveness of MT [3]. In current practice, the identification and selection of MRs are made manually, requiring a deep understanding of the SUT and its problem domain. The requirement for domain knowledge makes automatic MR identification challenging [3], [4]. Another critical challenge is the need to distinguish automatically whether a particular MR violation is due to a fault in the SUT or because the MR does not satisfy or fully fit a specific statement/method/function of the SUT for certain test data. In current practice, interpreting an MR violation is an entirely manual effort. It is important to highlight that the cost, in terms of time and resources, of the MT approach is related to the amount of MRs used [5], [6]. Thus, as the number of MRs grows, the number of test cases may grow exponentially. As a result, the execution time and the time needed for manual inspection of MR violations will also increase.

Some approaches indirectly reduce the manual effort required to interpret the meaning of an MR violation. For instance, Cao *et al.* [7] provides quantitative suggestions/guidance for developing automated means of selecting/prioritising MR for cost-effective MT. Srinivasan *et al.* [5], [6] proposed two MR prioritisation approaches to improve MT's efficiency and effectiveness. These approaches use (i) fault detection information and (ii) statement/branch coverage information to prioritise MRs. Zhang *et al.* [8] suggested strategies to clean MRs by deleting duplicate or redundant MRs. These approaches offer indirect help since by prioritising or reducing the set of MRs, the number of test cases will be reduced as well. Thus, the manual effort of inspection through the violated MRs is less.

Motivated by the above, we ask ourselves the following research question: *How can MRs be refined based on test data?*. To answer this question, we developed a method for refining MRs that suggest whether a detected MR violation results from a fault in the SUT or arises from the fact that the MR does not apply to the used test data. Our method assumes that a predefined set of MRs is provided and uses the concepts of fuzz testing, passive testing, and rule mining.

First, our method uses a fuzzer to feed random data to the SUT. Second, it performs the necessary input transformations following the indications of the MRs. Third, similar to passive tests, logs are produced with information related to inputs, outputs, and whether or not MRs are violated. Those logs are used to feed a mining algorithm. Our method employs

association rule mining (ARM). In our context, the purpose of ARM is to extract interesting relationships between the inputs and whether or not the MR is violated. ARM is an unsupervised machine learning (ML) method [9]. ARM algorithms attempt to find relationships or associations between categorical variables in large transactional datasets [10]. We were particularly interested in understanding whether the information provided by the resulting model helps in deciding whether there is a fault or whether the MR does not fully fit the specific method/function/statement when the violation occurs.

In our initial proof-of-concept, we validate our method on a toy example and discuss the lessons learned from our experiments. Our proof-of-concept demonstrates that our method is applicable and can provide suggestions that help strengthen the test suite for regression testing purposes. We published the replication package to facilitate future research.

The rest of the paper is structured as follows. Section II presents the main concepts used in our research. In Section III, we describe the proposed method. In Section IV, we present our results and discuss threats to validity. Section V presents the related work. Finally, we conclude the paper in Section VI.

II. BACKGROUND

This section presents the key concepts used in our research. Section II-A introduces the MT approach. Section II-B provides a brief description of test data generation techniques, and Section II-C gives a brief introduction to ARM.

A. Metamorphic Testing

MT is a software testing approach that alleviates the test oracle problem. MT aims to exploit the internal properties of a SUT to either check its expected outputs or generate new test cases. Figure 1 shows the MT basic workflow. Overall, MT works by checking the relation between the inputs and outputs of multiple executions of the SUT. Such relations are called MRs. MRs specifies how the outputs should change according to specific variations made to the input. Overall, MT follows four major steps:

- 1) Create a set of initial tests or source test cases.
- Identify an appropriate list of MRs that the SUT should satisfy.
- 3) Create follow-up test cases by applying the input transformations required by the identified MRs in Step 2 to each source test case.
- 4) Execute the corresponding initial and follow-up test case pairs.
- 5) Check if the source tests and follow-up tests output change matches the change predicted by the MR.

The final step needs further interpretation of the MT workflow output based on the Non-violation of MRs. When a Nonviolation is presented, it is not guaranteed that the SUT is implemented correctly. However, if an MR is violated for specific test cases, it must be a fault in the SUT, assuming the MR is defined correctly.



Fig. 1: MT basic workflow

B. Test Data Generation

In software testing, test data corresponds to the input data used during test execution. Test data is used for positive testing, verifying that functions of the SUT generate anticipated outputs for given inputs, and negative testing, examining the SUT's capacity to handle atypical, extraordinary, or unexpected inputs [ref]. Inadequately constructed testing data could only cover some potential test cases, which would be detrimental to the software's quality.

Our method does not attempt to add something new in test data generation techniques. Instead, we take advantage of Fuzz Testing for generating test data. Fuzz Testing, often known as "fuzzing," is a software testing approach involving injecting erroneous or random data, or "FUZZ," into software systems to find coding errors and security flaws. Fuzz testing involves injecting data using automated or somewhat automated methods and evaluating the system for various exceptions, such as system failure or malfunction of built-in code, etc [11].

Overall, fuzzing consists of three components, i.e., *input* generator, executor, and defect monitor [11]. The input generator provides the executor with several inputs, and the executor runs target programs on the inputs. Then, fuzzing monitors the execution to check if it discovers new execution states or defects (e.g., crashes). Fuzzing can be divided into Generation-based and Mutation-based fuzzing. Mutation-based fuzzing alters existing data samples to create new test data. Generation-Based fuzzing defines new data based on the system's input or the target SUT function. It starts generating input from scratch based on the specification.

C. Association Rule Mining

ARM is a rule-based unsupervised ML method that allows discovery relations between variables or items in large databases. ARM has been used in other fields, such as business analysis, medical diagnosis, and census data, to find out patterns previously unknown [10]. The ARM process consists of at least two major steps: finding all the frequent itemsets that satisfy minimum support thresholds and generating strong association rules from the frequently derived itemsets by applying a minimum confidence threshold.

A large variety of ARM algorithms exist. [12]. In our experiments, we use the Apriori algorithm from Python3 Efficient-Apriori library [13]. It is well known that the Apriori



Fig. 2: Overview of the method for refining MRs based on rule mining

algorithm is exhaustive; it finds all the rules with specified support and confidence. In addition, ARM doesn't require labelled data and is, thus, fully unsupervised. Below we define important terminology regarding ARM:

Itemset: Let X_i be items, then $I = \{X_1, ..., X_k\}$ is an Itemset of k different items, with k > 1.

Association rule: Consider a dataset D, having m different types of items and n transactions defined by the itemsets constructed from the items. An association rule exposes the relationships between the elements of the itemsets in the set of n transactions.

Support: The support of an association rule involving itemsets X and Y is the percentage of transactions in dataset D that contain itemsets X and Y. The support of an association rule $X \rightarrow Y$:

 $support(X \to Y) = support(X \cup Y) = P(X \cup Y)$

Confidence: The confidence is the percentage of transactions in the dataset D with itemset X that also contains the itemset Y. The confidence is calculated using the conditional probability, which is further expressed in terms of itemset support: $confidence(X \rightarrow Y) = P(Y|X) = support(X \cup Y)/support(X)$

Lift: Lift is used to measure the frequency of the occurrence of X and Y together if both are statistically independent of each other. The lift of rule $(X \rightarrow Y)$ is defined as $lift(X \rightarrow Y) = confidence(X \rightarrow Y)/support(Y)$.

A lift value of 1 indicates that X and Y appear as frequently together as they appear individually under the assumption of conditional independence.

III. METHOD

Figure 2 presents an overview of the method for refining MRs based on rule mining. In general, the proposed method comprises two phases. Phase I is in charge of identifying the degree of applicability of the set of MRs selected. It is important to highlight that our method does not cover the selection of appropriate MRs. We assume that there is already a predefined set of several MRs. Phase II uses the refined MRs to create or improve the test suite for future SUT versions.

Thus, the output of this phase could be seen as a regression test suite. Each phase is thoroughly described below:

A. Phase I

Phase I comprises three modules: Test Data Generation Module (TDG Module), Metamorphic Test Module (MT Module) and Analyser Module. Overall, the TDG Module produces the test data that will feed the SUT and the MT Module. In the MT Module, the test data is transformed based on the indications of each MR; then, such transformed data is executed against the SUT. Both SUT outputs, the output produced with test data and the transformed test data, are executed against SUT, are checked against the MRs in the MR Checker. Then, the test data and the results of the MR Checker Module are organised and stored in a Log file. The Log file is used in the Analyser Module, where processed and refined MRs based on rules are extracted. These modules, their internal settings and their activities are detailed below:

1) TDG Module: This module generates the test data, which will feed the SUT. It is important to note that our method does not try to add something new in the field of test data generation techniques. As we explained in Section II-B, our method uses fuzzing testing to generate test data. Overall, the basic fuzzing workflow involves three basic components, input generator, executor, and defect monitoring. There are open-source tools that can be used in this module. However, it is necessary to consider the SUT's application domain. For instance, fuzzers such as SPIKE proxy, Peach Fuzzer, and OWASP WSFuzzer are highly recommended for security purposes in web systems. Also, the programming language must be considered when selecting the fuzzer. For instance, OSS-Fuzz, Google's open-source fuzzing platform, supports Java and Python language for finding security vulnerabilities, stability issues, and functional bugs. Regardless of the Fuzzer used, this module's most important is storing the generated test data.

2) *MT Module:* Overall, this module is responsible for performing the test data transformation based on the changes in the inputs specified by the MRs, executing the transformed

test data, and checking if the test data and the transformed test data outputs match the change predicted by the MRs. This module has three main activities, *Set of MRs, Test data transformations*, and *MR Checker*.

- Set of MRs: It is important to note that our approach does not involve the initial selection of the MRs. Our approach assumes the prior existence of a predefined set of MRs.
- *Test data transformation:* This activity is in charge of transforming the test data according to the change specified by each MR. To the best of our knowledge, no tool performs this activity automatically, *i.e.*, the translation of input change described by the MR and its meaning into code. Thus, this is considered to be a manual task.
- *MR Checker:* This activity is responsible for checking that both outputs, test data and transformed test data match the change predicted by the corresponding MR.

Once the test data is generated, transformed, and compared by the MR Checker, *i.e.*, the verdict of the MR Checker is ready, a Log file is produced with the following information: execution ID, test data, function call, and the MR Checker verdict per MR.

3) Analyser Module: The Analyzer Module is in charge of discovering interesting relations between test data and whether or not a certain MR has been violated. This module has three main activities, *Pre-processing*, *Tester feedback*, and *Rule mining*. Below we describe each activity in detail as well as their internal process:

- *Pre-processing:* This activity is responsible for ensuring that the data is correct, consistent and usable. Also, it shows the tester an initial summary of the percentage of violations and not violations per each MR and function call. This activity has three main functions: *data quality*, *clean*, and *summary report*. The *data quality* function is responsible for checking that Log has no missing data, as well as removing the rows that are not needed or inconsistent rows. The *clean* removes duplicate entries. The *Summary report* is based on the percentage of violations and no-violation per MRs. Also, it provides the tester with the ability to inspect some random sample values and atypical values. This is done to increase confidence in the suitability of the MRs.
- *Tester feedback:* This activity is in charge of qualifying MR's verdict based on the summary report provided by the *Pre-processing* activity. Here the tester needs to perform two checks when there is an incorrect and correct behaviour of the MR Checker output:
 - a) *Incorrect behaviour*: The tester evaluates whether there is an obvious fault in the SUT. Phase I should be repeated as long the fault is present.
 - b) *Correct behaviour*: In the correct behaviour there are two possibilities.
 - b.1) MR not violated which represents a positive test.

b.2) MR violated which represents a negative test.

If specific MR is violated 100% of the time, we assume that the MR does not apply to the tested function. On

the other hand, if it is not violated 100% of the time, we assume that the MR matches the tested function. In both cases, the tester can directly decide whether to include them in the Rule file. Those 100% violations could be used as negative tests and the 100% of no violations as positive tests. We also look at the atypical values; for instance, if the MR was violated or not violated only 10% of the time.

- *Rule mining (predefined data type relations):* This activity is responsible for generating the rule set by discovering interesting relationships between the test data and whether or not a particular MR has been violated. This activity has three internal steps:
 - 1) *Encoding:* This step is in charge of preparing the data according to the requirements of the rule mining algorithm. For example, Apriori [14], which is the algorithm used in this paper, works only with categorical features. Thus, this component categorises and generalises the numerical inputs into string representations.
 - 2) *Rule generation:* This step is responsible for generating the set of rules using the Apriori ARM algorithm.
 - 3) Data type relation: This step is in charge of generalising the data based on its data type relation. This data type is predefined. For example, the test data for SUT can be generalised using partial order theory. The partial order defines a notion of comparison between at least two elements, *e.g.*, input_a and input_b. The two elements input_a and input_b can be in any of four mutually exclusive relationships to each other:

input_a < input_b input_a = input_b input_a > input_b input_a and input_b are incomparable

The latter relationship is not present in our data.

B. Phase II

Phase II is in charge of generating the test suite for regression testing purposes. Overall, this phase takes the Log, which has the test data generated and takes the Log with the set of rules for generating the final test suite.

IV. RESULTS AND DISCUSSION

This section presents and discusses the preliminary results of our approach. Let's consider the Algorithm 1 as the SUT. Algorithm 1 is a program that computes three basic arithmetic operations, addition, subtraction, and multiplication between two integers. The full set of data generated during our experiments, as well as all scripts, can be found in our GitHub repo¹.

A. Phase I

1) **Test Data Generation Module:** In this module, we create a fuzzer based on a random number generator. For this,

¹https://github.com/aduquet/VST2023-BugORNOTbug

Alg	Algorithm 1 Calculator()			
1:	procedure ADDSUB(int a, int b)			
2:	function ADD(a, b)			
3:	return a + b			
4:	function SUB(a, b)			
5:	return a - b			
6:	function MUL(a, b)			
7:	return a * b			

we use the NumPy random function in Python. A total of 100 random numbers, elements of $\{0, 1, 2, ..., 9\}$, were generated for input_a and input_b, following a uniform distribution. Since, for some MRs, a constant is needed, it was generated only once and reused every time it was needed. Figure 3 shows the histogram of the generated test data, *i.e.*, input_a and input_b.



Fig. 3: Distribution of data generated for $input_a$ and $input_b$

2) MT Module:

• Set of MRs: Table I describes the set of MRs for Algorithm 1. For our SUT, we take advantage of the generic rules of arithmetic to build a set of four MRs that may apply to the SUT. MR₂, MR₃, and MR₄ need a constant k. That constant was randomly generated only once and reused each time it was needed.

TABLE I: Set of MRs for Algorithm 1

MR	Change in the input	Expected output
MR1	Permute the inputs	Remain equal
MR2	Multiply by a positive constant $k > 1$	Increase
MR3	Adding a positive constant k to each operand	Remain equal
MR4	Subtracting a positive constant k from each operand	Remain equal

- *Test data transformation* Table II shows how the inputs are transformed following the MR indications.
- *MR Checker:* Table III shows how the MR Checker checks the outputs following the expected output predicted by the MRs and gets the verdict, *i.e.*, not violated or violated for our SUT.

3) Analyser Module:

- *Pre-procesing:* Figure 4 shows an example of the summary report for Algorithm 1, *i.e.*, not violated or violated, with a controlled input space.
- Tester feedback: Figure 4 shows the percentage of MR • not violated and violated per function call. From the first line, which belongs to the ADD function, we can see that for MR₃ and MR₄, there are 100% violations. In this case, we assume that MR3 and MR4 do not apply to the SUM function. However, we have two options: include a test (negative test) for all test data or not include it. on the other hand, MR1 and MR2 present 100% and 99% of not violation respectively. After inspecting random samples to check, we can conclude that MR₁ and MR₂ completely match the ADD function. This means that MR1 and 99% of MR₂ will be directly included in the set of rules to build positive tests in phase II. The 1% violation is a particular case for MR₂ in the ADD function, which occurs whenever input_a and input_b are equal to 0.



Fig. 4: Summary report for the tester feedback

In the second line of pie charts in the Figure 4, which belongs to the SUB function, you can see the opposite behaviour of MR_3 and MR_4 compared to the ADD function. For these MR we have 100% non-violations. This indicates that the MRs fully match the SUB function. Like MR_1 and MR_2 in the ADD function, MR_3 and MR_4 can go directly as a positive test for the SUB function in phase II. We can't say to much about MR_1 and MR_2 , are the ones who we are interested to analyse with the *rule mining*.

The results from MUL function are in the third line of the pie charts in the Figure 4. In this function, one can see that MR_1 and MR_3 behave the same as in the ADD function, which means that MUL also fully matches MR_1 and can be included directly in the final rules. It similarly

TABLE II: Test data transformations of the test data according with the MRs described in Table I for Algorithm 1

MR	Input change	Test data (input _a , input _b)	transformed test data $(T-input_a, T-input_b)$	
\mathbf{MR}_1	Permute the inputs		$T\text{-input}_a = b, T\text{-input}_b = a$	
MR_2	Multiply by a positive constant $k > 1$	†:	T-input _a = a * k , T -input _b = b * k	
MR_3	Adding a positive constant k to each operand	$\operatorname{Input}_a = a, \operatorname{Input}_b = b$	T-input _a = a + k , T-input _b = b + k	
MR_4	Subtracting a positive constant k from each operand		$T\text{-input}_a = a - k$, $T\text{-input}_b = b - k$	
† a, b are elements of {0, 1, 2,, 9}				

TABLE III: MR Checker verdict example according with the expected outputs predicted by the MRs described in Table I for Algorithm 1

MR ₁ Remain equal t_{iarrut} = b, T-input _a = b, T-input _b = a SomeFunc(i	$put_a, input_b) == SomeFunc(T-input_a, T-input_b)$
MR ₂ Increase T-input _a = a * k , T-input _b = b * k SomeFunc(i	$\operatorname{put}_a,\operatorname{input}_b) < \operatorname{SomeFunc}(\operatorname{T-input}_a, \operatorname{T-input}_b)$
MR ₃ Remain equal $t_{input_a} = b$ T-input _a = a + k, T-input _b = b + k SomeFunc(i	$\operatorname{SomeFunc}(\operatorname{T-input}_a, \operatorname{T-input}_b) = \operatorname{SomeFunc}(\operatorname{T-input}_a, \operatorname{T-input}_b)$
$\mathbf{MR}_4 \text{Remain equal} \text{Imput}_b = b \text{T-input}_a = a - k \text{, T-input}_b = b - k SomeFunc(integration of the second $	$\operatorname{put}_a,\operatorname{input}_b) == \operatorname{SomeFunc}(\operatorname{T-input}_a,\operatorname{T-input}_b)$

[†] a, b are elements of {0, 1, 2, ..., 9}

SomeFunc refers to a function call of the class, i.e., ADD, SUB or MUL in Algorithm 1

TABLE IV: MR Checker output example

id	$(input_a, input_b)$	Func Call	MR ₁	MR ₂	MR ₃	MR_4
0	(0,0)	add	✓	X	X	X
1	(1,1)	sub	\checkmark	X	X	X
2	(2,1)	mul	\checkmark	X	×	X
299	(9,9)	add	✓	√	X	X

✓: Not violate, X: Violated

happens with MR₃; it has the same behaviour as the ADD function, meaning we can treat it in the same way as MR₃ for ADD function. The 11% of the violated cases in MUL function for MR₂ occurs whenever input_a or input_b are equal to 0. Here one could assume that we have other particular case in the test data. It indicates that we can use the this particular case to create a negative MR₂ test for MUL. It cannot be the same as in the ADD function since in the MUL function, these violations occur when either input_a or input_b is 0, and in ADD function, both inputs must be equal to zero. From the information in Figure 4 one can assume the following:

- a) MR₁ applies to ADD and MUL function for all the test data.
- b) MR_2 applies to the SUB function for almost all the test data, but when $input_a == input_b == 0$, MR_2 must be used as a negative test.
- c) MR_3 and MR_4 does not apply to the ADD function.
- d) MR₃ and MR₄ applies to the SUB function for all the test data.
- f) MR_3 does not apply to the MUL function.
- *Rule mining:* We apply the Apriori algorithm with minimal support and maximum confidence thresholds, *i.e.*, 0.2 and 1. We created the data type relation in the following way:

$$input_a < input_b$$

 $input_a = input_b$

$input_a > input_b$

Table V shows the final set of detected rules. The first three rules are the ones that were directly inferred from the Tester Feedback. The other three rules are the output of the ARM. Table V shows that the relation a < b has the largest support. This is due to the amount of samples that fall in that range. The lower support belongs to the relation a == b, meaning that not many samples fall into that range.

TABLE V: Final set of rules

RHS	LHS	conf	sup	lift
('a == b == 0', ADD)	$MR_2 = Violated$			
('a == b', SUB)	MR ₂ = Not violated			
(a == 0' or b == 0', MUL)	MR2 = Violated			
('a < b', SUB)	$MR_2 = Violated$	1.0	0.46	1.639
('a > b', SUB)	$MR_2 = Not violated$	1.0	0.32	2.564
('a == b', SUB)	$MR_2 = Violated$	1.0	0.06	1.639
DHC D' L H LO'L THC L	C II 1 C'1			

RHS: Right Hand Side, LHS: Left Hand Side

B. Phase II

With the information assumed in the feedback tester step and the Table V, one can create a test suite for regression testing purposes. Figure 5 shows an example of test code for the SUB function using MR_1 and the ADD function using MR_2 . The full example can be found in our GitHub repository.

C. Remarks on effectiveness

In terms of MRs selection, our method cannot be worse than any existing method that manually create MRs. First, our method reuses the already created MRs. Second, our method can add more MRs by refining the existing ones with specific test data. Regarding the inspection of the truthfulness of the test output, *i.e.*, whether or not the MR is violated, it is important to note that using our approach, whatever the results of throwing data against the set of MRs, it will always be the same situation if the MRs have been manually created. In

```
from toy example import calculator
from unittest import TestCase
class AddSubTest(TestCase):
   global a, b, constant
    a = [0, 2, 3, 2, 0]
   b = [1,2,9,1,2]
    constant = 9
   def test sub MR1(self):
        for i in range(0,len(a)):
            if a[i] == b[i] == 0 or b[i] == a[i]:
                expected = calculator(b[i],a[i]).subtraction()
                self.assertEqual(calculator(a[i],b[i]).subtraction(), expected)
            else:
                expected = calculator(b[i],a[i]).subtraction()
                self.assertNotEqual(calculator(a[i],b[i]).subtraction(), expected)
   def test_add_MR2(self):
        # Output must to be <
        for i in range(0,len(a)):
            # Violation case
            if a[i] == 0 and b[i] == 0 :
                expected = calculator(constant * a[i], constant * b[i]).add()
                self.assertEqual(calculator(a[i],b[i]).add(), expected)
            #No-violation case
                expected = calculator(constant * a[i], constant * b[i]).add()
                self.assertLess(calculator(a[i],b[i]).add(), expected)
```

Fig. 5: Example test suit using refined MRs

fact, it is part of the MT approach to inspect such results, *i.e.*, whether or not the MR is violated.

The advantage of our approach is that it can provide hard facts on which MRs out of the set of MRs can and cannot be applied. In this regard, our approach reduces the time spent in selecting MRs, which is a manual and very time-consuming task. The downside of our approach is the time required for setup, the runtime of MRs against the test data, and manual inspection. However, the execution of the MRs with the test data can be automated, and the manual inspection is performed only once.

In terms of the effectiveness of our approach to finding defects, we face the same problem as any testing approach. It is well known that there is an open and old discussion about the dependency of test data and test cases. Any approach that uses MR can only find fault with them if the test data is chosen wisely. The proper selection of test data is planned to be explored in the future.

D. Threats to validity

In the context of our proof-of-concept validation, two types of threats to validity are most relevant: threats to internal and external validity.

1) Internal validity: We used a well-known program as a proof-of-concept and the most common ARM algorithm to achieve internal validity. It is not fully clear in which situations the ARM is the best choice for our method. Future research in the direction of evaluating different ARM algorithms and feature selection needs to be done.

2) *External validity:* With regard to external validity, our study is rather limited since we only use one well-understood class in our experiments. Thus, the actual scope of the effectiveness of our proposed method is yet to be determined.

3) Construct validity: In this paper, we used the NumPy package, in particular its random function, to generate the test data. The usage of these third-party libraries represents potential threats to construct validity. To avoid this, we verified that the results produced by the random function are uniform by manually inspecting the generated distributions.

V. RELATED WORK

Since MT was introduced in 1998 by Chen *et al.* [2], MT has been demonstrated to be an effective technique for testing in a variety of application domains. Several studies have shown MT as a strong technique for testing the "non-testable programs" where an oracle is unavailable or too difficult to implement [15]–[18]. Also, MT has been demonstrated to be an effective technique for testing in a variety of application domains, *e.g.*, autonomous driving [19], [20], cloud and networking systems [21], [22], bioinformatic software [23], [24], scientific software [25]. However, the efficacy of MT heavily relies on the specific MRs employed and its interpretation of the meaning of MR violations.

As we mentioned before, some approaches indirectly reduce the manual effort required to interpret the meaning of an MR violation through prioritisation. For instance, Cao *et al.* [7] provides quantitative suggestions/guidance for developing automated means of selecting/prioritising MR for cost-effective MT. Srinivasan *et al.* [5], [6] proposed two MR prioritisation approaches to improve MT's efficiency and effectiveness. These approaches use (i) fault detection information and (ii) statement/branch coverage information to prioritise MRs. Zhang *et al.* [8] suggested strategies to clean MRs by deleting duplicate or redundant MRs. These approaches offer indirect help since by prioritising or reducing the set of MRs, the number of test cases will be reduced as well. Thus, the manual effort of inspection through the violated MRs is less.

VI. CONCLUSION AND FUTURE WORK

We presented a new ARM-based method for refining MRs that suggest whether a detected MR violation results from a fault in the SUT or arises from the fact that the MR does not apply for the used test data. Our method assumes that a predefined set of MRs is provided and uses the concepts of fuzz testing and ARM. Our method consists of two phases. The purpose of Phase I is to evaluate the level of applicability of the chosen set of MRs.

In phase I, there are three main modules: TDG Module, MT Module, and Analyser Module. The TDG Module generates the test data that will be sent to the MT Module and SUT. According to each MR's instructions, the test data is changed in the MT Module before being run against the SUT. In the MR Checker, the output from running test data and the transformed test data against SUT are compared to the changes predicted by the MRs. The test data and the MR Checker Module's results are then organised and saved in a Log file. The Analyser Module uses the Log file to process and improve MRs according to relations test data and whether or not the MR is violated. Phase II is in charge of analysing the final set of rules and creating the new test suite. In our proof-of-concept, we used a toy example, a program that computes three basic arithmetic operations, addition, subtraction, and multiplication between two integers. We show step by step the execution of our method and its expected outputs from each module.

An advantage of our method is that it can be applied not on the SUT with only integer inputs and outputs but on the class under test where we can have any inputs. Also, of mixed types of data. Thus, our method can be generalised for inputs of any type, not only for integers. It removes some limitations on the type of SUT that can be analysed. The weakness of our method is the need for manual feedback from the tester. However, compared with the manual effort already needed in the MT approach, we consider that the effort needed in our approach is less. We must manually translate the rules into assertions (test code) to generate the final test suite. For efficiency reasons, it would be better to have an automatic translation of the rules generated into test code. Unfortunately, there is no simple way to do this.

Given the limitations of our study, more experiments have to be conducted to test our proposed method empirically. We are currently focusing on extending our experiments in three directions. First, we will add more MRs in the initial set of MRs to test the sensitivity of our method with regard to the filtering MRs that have no relation to the SUT. We will systematise this by using the relation between the input type and the MR. Second, we will apply our proposed method to more SUT. Third, we will do a mutation analysis to evaluate the effectiveness of our approach.

ACKNOWLEDGEMENT

This research was partly funded by the Estonian Center of Excellence in ICT research (EXCITE), the IT Academy Programme for ICT Research Development, the Austrian ministries BMVIT and BMDW, the Province of Upper Austria in the frame of the Software Competence Center Hagenberg (SCCH), and grant PRG1226 of the Estonian Research Council.

REFERENCES

- A. Duque-Torres, D. Pfahl, A. Shalygina, and R. Ramler, "Using rule mining for automatic test oracle generation," in 8th International Workshop on Quantitative Approaches to Software Quality (QuA-SoQ@APSEC), 2020.
- [2] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," *Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep. HKUST-CS98-01*, 1998.
- [3] A. Duque-Torres, D. Pfahl, R. Ramler, and C. Klammer, "A replication study on predicting metamorphic relations at unit testing level," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 709–719.
- [4] A. Duque-Torres, D. Pfahl, C. Klammer, and S. Fischer, "Using source code metrics for predicting metamorphic relations at method level," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 1147–1154.

- [5] M. Srinivasan and U. Kanewala, "Metamorphic relation prioritization for effective regression testing," *Software Testing, Verification and Reliability*, vol. 32, no. 3, e1807, 2022.
 [6] M. Srinivasan, "Prioritization of metamorphic relations based on test
- [6] M. Srinivasan, "Prioritization of metamorphic relations based on test case execution properties," in 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, pp. 162–165.
- [7] Y. Cao, Z. Q. Zhou, and T. Y. Chen, "On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions," in 2013 13th International Conference on Quality Software, 2013, pp. 153–162.
- [8] 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.
- [9] L. Zhou and S. Yau, "Efficient association rule mining among both frequent and infrequent items," *Computers and Mathematics with Applications*, vol. 54, no. 6, pp. 737–749, 2007.
- [10] S. K. Solanki and J. T. Patel, "A survey on association rule mining," in 2015 Fifth Int'l Conf. on Advanced Computing Communication Technologies, 2015, pp. 212–216.
- [11] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," ACM Comput. Surv., vol. 54, no. 11s, Sep. 2022.
- [12] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, ser. VLDB '94, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.
- [13] G. McCluskey, "Efficient-apriori documentation," 2018.
- [14] A. Bhandari, A. Gupta, and D. Das, "Improvised apriori algorithm using frequent pattern tree for real time applications in data mining," *Procedia Computer Science*, vol. 46, pp. 644–651, 2015.
- [15] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.
- [16] 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, Jan. 2018.
- [17] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *In Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2008.
- [18] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing: Testing the untestable," *IEEE Software*, vol. 37, no. 3, pp. 46– 53, 2020.
- [19] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 132–142.
- [20] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Communications of the ACM*, vol. 62, no. 3, pp. 61–67, Feb. 2019.
- [21] P. C. Canizares, A. Núnez, J. de Lara, and L. Llana, "MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems," *Journal of Systems and Software*, vol. 163, p. 110 522, 2020.
- [22] Z. Zhang, D. Towey, Z. Ying, Y. Zhang, and Z. Q. Zhou, "MT4NS: Metamorphic testing for network scanning," in 6th IEEE/ACM International Workshop on Metamorphic Testing (MET), ser. MET'21, 2021, pp. 17–23.
- [23] M. Srinivasan, M. P. Shahri, I. Kahanda, and U. Kanewala, "Quality assurance of bioinformatics software: A case study of testing a biomedical text processing tool using metamorphic testing," in *IEEE/ACM 3rd International Workshop on Metamorphic Testing* (*MET*), ser. MET'18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 26–33.
- [24] M. P. Shahri, M. Srinivasan, G. Reynolds, D. Bimczok, I. Kahanda, and U. Kanewala, "Metamorphic testing for quality assurance of protein function prediction tools," in *IEEE International Conference* On Artificial Intelligence Testing (AITest), IEEE, 2019, pp. 140–148.
- [25] Z. Peng, U. Kanewala, and N. Niu, "Contextual understanding and improvement of metamorphic testing in scientific software development," in 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2021, pp. 1–6.