DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
63

ALEJANDRA DUQUE TORRES

Classifying, Constraining and Ranking Metamorphic Relations

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
63

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
63

ALEJANDRA DUQUE TORRES

Classifying, Constraining and Ranking Metamorphic Relations

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on January 7, 2025 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor

Prof.	Dietmar Alfred Paul Kurt Pfahl University of Tartu, Estonia

Opponents

Prof.	Shaukat Ali Simula Research Laboratory, Norway
Assoc. Prof.	Emelie Engström Lund University, Sweden

The public defense will take place on February 6, 2025 at 10:15 in Narva Rd.18-1019.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (pdf)
ISBN 978-9916-27-785-0 (print) ISBN 978-9916-27-786-7 (pdf)

University of Tartu Press
`http://www.tyk.ee/`

To my family and friends

ABSTRACT

Software testing is a crucial part of the software development process, ensuring that the final product operates as expected. However, it is often a time-consuming, resource-intensive, and complex activity, particularly for large-scale systems. Automation is employed to reduce costs and improve efficiency by automating the execution of test suites. Despite advancements, generating effective Test Data (TD) and determining the correct output remain major challenges in software testing, with the latter commonly referred to as the test oracle problem. The test oracle problem arises when the SUT lacks an oracle or when developing one to verify the computed outputs is practically impossible. Addressing the test oracle problem has been the focus of significant research, and while some progress has been made through model-driven testing, it remains largely unsolved.

Metamorphic Testing (MT) is a software testing approach to alleviate the test oracle problem. Unlike traditional testing techniques, MT analyses the relations between pairs of input-output combinations across consecutive executions of the SUT rather than focusing solely on verifying individual input-output combinations. Such relations between SUT inputs and outputs are known as Metamorphic Relations (MRs). MRs specify how the outputs should vary in response to specific input changes. When an MR is violated for at least one valid test input, it indicates a high probability of a fault within the SUT. Nevertheless, the absence of MR violations does not guarantee a fault-free SUT. However, its effectiveness heavily relies on the MRs employed. A good MR must not only specify correctly the input-output relations across the valid input data space - it also must be capable of detecting incorrect program behaviour resulting from a fault in the SUT code.

Generating good MRs is not straightforward. Approaches to MR generation fall into two categories: semantic and syntactic correctness. Semantically correct MRs require an in-depth understanding of the SUT's behaviour, often based on software documentation and expert knowledge. These are typically created manually, which is time-consuming and requires deep domain expertise. On the other hand, syntactically correct MRs focus on ensuring that the MR adheres to the input-output structure of the SUT. These MRs are often generated through automated methods based on predefined patterns or generic MRs applicable across multiple systems. However, even with these approaches, selecting and classifying effective MRs remains a challenge. Several methods, such as the Predicting Metamorphic Relation (PMR) approach, have been proposed to automate MR classification. PMR uses Machine Learning (ML) to classify MRs based on code structure, typically through Control Flow Graph (CFG) or source code metrics.

While these methods have shown promising results, they face notable limitations. Firstly, it relies on binary classifiers that require labelled datasets to provide examples for learning. Labelled datasets may not always be available, and obtaining them can be time-consuming. Secondly, the feature extraction process for

model training is based on CFG or source code metrics, which may not account for refactoring. This limitation can affect the accuracy of the PMR approach, as refactoring can change the structure of the code and, consequently, the way MRs apply. Lastly, the binary output of PMR may not consider TD and its impact on MR applicability. This limitation implies that PMR does not consider the possibility that an MR may apply to some TD with specific characteristics and not others, leading to false positives or false negatives in MR selection. Given these challenges, the goal of this thesis is to introduce new methods for classifying, refining, and assessing MRs to improve the efficiency and effectiveness of MT.

The thesis contributions are divided into three parts. The first contribution, MetaTrimmer, introduces a novel TD-driven method for classifying MRs. Unlike traditional classification methods based on static code structure, MetaTrimmer dynamically evaluates MR behaviour across different TD inputs. By incorporating the behaviour of TD, MetaTrimmer moves beyond the assumption of universal applicability, allowing it to classify MRs more accurately by identifying specific TD subsets where MRs are valid. The second contribution, MetaTrimmer+, extends the analysis and refinement of MRs, particularly for mixed-case MRs—those that may exhibit both violations and non-violations depending on the TD. MetaTrimmer+ uses a combination of manual inspection and Association Rule Mining (ARM) to extract patterns from the TD space, clarifying when MRs behave as always not violated or always violated. This refinement enhances the usability of mixed MRs, allowing them to be applied as both positive and negative test cases, which expands the test suite and improves its effectiveness. The third contribution focuses on assessing the strength of MRs based on their defect-detection capabilities. By integrating MetaTrimmer with mutation testing, the thesis presents a method for ranking MRs according to their effectiveness in finding bugs. This allows testers to prioritise the most effective MRs, optimising the test suite by reducing its size without compromising its ability to detect defects. The method was successfully applied in an industrial case study, demonstrating its practical value, although the computational cost of mutation testing and reliance on diverse TD remain challenges.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Acronyms

ARM Association Rule Mining. 7, 27–30, 65

CFG Control Flow Graph. 6, 12, 15, 24, 35–39, 42, 47–50, 137

GK Gaussian Kernel. 47, 50

GMRs Generic MRs. 80

HMRs Handcrafted MRs. 80

LLMs Large Language Models. 23

ML Machine Learning. 6, 24, 140

MRs Metamorphic Relations. 6, 12, 21–31, 47, 48, 79, 129

MT Metamorphic Testing. 6, 21, 22, 24, 25, 28, 30–32, 48, 110

OAs Optimisation Algorithms. 79, 80

PMR Predicting Metamorphic Relation. 6, 8, 11, 12, 24–26, 31, 34, 35, 37, 38, 47–50, 129, 133, 134, 137, 141, 146

RGs Research Goals. 25, 30

RQs Research Questions. 25, 30

RWK Random-Walk Kernel. 47, 48, 50

SUT System Under Test. 21–24, 26, 27, 29, 31, 32, 47, 79, 81

SVM Support Vector Machine. 46–48, 50

TD Test Data. 6–8, 21, 26–32, 48, 51, 52, 54–61, 63–65, 77, 78, 106, 111, 113–115

TTD Transformed Test Data. 26, 31, 52, 55

1. LIST OF ORIGINAL PUBLICATIONS

Publications in the scope of the thesis

I **A. Duque-Torres**, D. Pfahl, R. Ramler and C. Klammer, *"A Replication Study on Predicting Metamorphic Relations at Unit Testing Level,"* IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 709-719.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SANER53432.2022.00088

II **A. Duque-Torres**, D. Pfahl, C. Klammer and S. Fischer, *"Using Source Code Metrics for Predicting Metamorphic Relations at Method Level,"* IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 1147-1154.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SANER53432.2022.00132

III **A. Duque-Torres** and D. Pfahl, *"Inferring Metamorphic Relations from JavaDocs: A Deep Dive into the MeMo Approach,"* in Product-Focused Software Process Improvement (PROFES), Springer International Publishing, 2022, pp. 418–432.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1007/978-3-031-21388-5_29

IV **A. Duque-Torres**, D. Pfahl, C. Klammer and S. Fischer, *"Exploring a Test Data-Driven Method for Selecting and Constraining Metamorphic Relations,"* 49^{th} Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Durres, Albania, 2023, pp. 370-377.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SEAA60479.2023.00063

V **A. Duque-Torres**, D. Pfahl, C. Klammer and S. Fischer, *"Bug or Not Bug? Analysing the Reasons Behind Metamorphic Relation Violations,"* IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Taipa, Macao, 2023, pp. 905-912.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SANER56733.2023.00109

VI **A. Duque-Torres**, C. Klammer, S. Fischer, and D. Pfahl, *"Is It The Best Solution? Testing an Optimisation Algorithm with Metamorphic Testing,"* in Product-Focused Software Process Improvement (PROFES), Springer Na-

ture Switzerland, 2023, pp. 339–354.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1007/978-3-031-49266-2_23

VII **A. Duque-Torres**, D. Pfahl, C. Klammer and S. Fischer, *"MetaExploreX: A Visualisation Tool for Selecting and Constraining Metamorphic Relations,"* IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C), Rovaniemi, Finland, 2024, pp. 219-222.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SANER-C62648.2024.00036

VIII **A. Duque-Torres**, D. Pfahl, C. Klammer, S. Fischer and R. Ramler, *"The Metamorphic Lighthouse: Understanding the Input Data Space of Metamorphic Relations,"* 50^{th} Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Paris, France, 2024.
Lead author. The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SEAA64295.2024.00064

Publications out of the scope of the thesis

I A. Walchshofer, S. Fischer, A. Wöß, **A. Duque-Torres**, M. Löberbauer and G. Koll, *"Introducing a Linter in an Industrial Lua Code Base,"* 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C), Rovaniemi, Finland, 2024, pp. 191-198.
The author performed the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SANER-C62648.2024.00032

II **A. Duque-Torres**, C. Klammer, D. Pfahl, S. Fischer and R. Ramler, *"Towards Automatic Generation of Amplified Regression Test Oracles,"* 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Durres, Albania, 2023, pp. 332-339.
Lead author. The author performed part of the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/SEAA60479.2023.00058

III **A. Duque-Torres**, N. Doliashvili, D. Pfahl and R. Ramler, *"Predicting Survived and Killed Mutants,"* IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, 2020, pp. 274-283.
Lead author. The author performed part of the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 10.1109/ICSTW50294.2020.00053

IV **A. Duque-Torres**, D. Pfahl, A. Shalygina, and R. Ramler, *"Using rule mining for automatic test oracle generation,"* in 8^{th} International Workshop on Quantitative Approaches to Software Quality (QuASoQ@APSEC), vol. 2767, 2020, pp. 21–28.
Lead author. The author performed part of the implementation and the analysis of the experiments and contributed with the main ideas and the writing.
Link: doi: 0000-0001-8763-3457

2. INTRODUCTION

Software testing is an essential activity of quality assurance in software development process as it helps ensure the correct operation of the final software [1]. However, software testing has historically been recognised to be a time-consuming, challenging, and expensive activity given the size and complexity of large-scale software systems [2]. The cost involved in testing can be managed through test automation. Test automation refers to the writing of special programmes that are aimed at exposing failures in the System Under Test (SUT) and to using these programmes together with standard software solutions (test frameworks) to control the execution of test suites. It is possible to use test automation to improve test efficiency.

Software testing, whether automated or manual, involves four major steps: generating Test Data (TD), determining the expected output of the SUT when using the TD, obtaining the actual output when executing the SUT using the TD against the SUT, and comparing the expected output with the actual output to determine the test verdict (pass/fail) [3]. In these steps, there are two major challenges: finding effective TD inputs, *i.e.,* inputs that can trigger failures in SUT, and determining the correct output after the execution of the test cases. The second challenge refers to the test *oracle problem.* A test oracle is a mechanism that determines the correct output of the SUT for a given input [4]. The test oracle problem arises when SUT lacks an oracle or when creating one to verify the computed outputs is too costly or practically impossible [5]. Although substantial research has been conducted to provide test oracles automatically, aside from model-driven testing, the oracle problem remains largely unsolved.

To address the test oracle problem, Chen *et al.* [6] introduced Metamorphic Testing (MT). MT tackles the test oracle problem by delving into the internal properties of the SUT and evaluating how outputs should vary with specific input changes. The essence of MT lies in analysing the relations between inputs and outputs during multiple SUT executions; these relations are referred to as Metamorphic Relations (MRs). An MR consists of two parts: the *input transformation statement* part, which outlines how to modify a given input to create a new related input, and the *expected output changes* part, which defines the relation that the outputs must exhibit when subjected to these input modifications [7]. To assess the correctness of the SUT using MT, it is necessary to check whether the output relations defined by the MR hold for the outputs produced by both the transformed TD input and the non-transformed TD input. If these relations do not hold, it signifies a violation, indicating a substantial likelihood of a fault in the SUT. However, the absence of MR violations does not guarantee a fault-free SUT.

Figure 1 illustrates the workflows of both traditional testing and MT. Traditional testing involves generating TD, determining the expected output (oracle), executing the TD against the SUT, and comparing the actual output to the ex-

pected output. Metamorphic testing, on the other hand, includes transforming the initial TD based on MRs and comparing the outputs from both the original and transformed TD to verify these relations. Tools like fuzzers, which leverage fuzz testing techniques to automatically generate diverse and often random input data for uncovering errors and vulnerabilities, can be employed to produce varied and extensive TD, ensuring broader input coverage. However, it is important to note that TD generation is not limited to fuzz testing; other tools and techniques, such as symbolic execution, combinatorial testing, and model-based testing, can also be utilized to create comprehensive and diverse test data tailored to specific testing requirements.

MT has been demonstrated to be an effective technique for testing in a variety of systems from different application domains, including autonomous driving [8], [9], optimisation process [10], [11], cloud and networking systems [12], [13], bio-informatics software [14], [15], web systems [16], [17], cyber-physical systems [18], [19], energy testing [20] and scientific software [21], [22]. However, it is well known that the effectiveness of MT highly depends on how *"good"* the MRs used are [6].

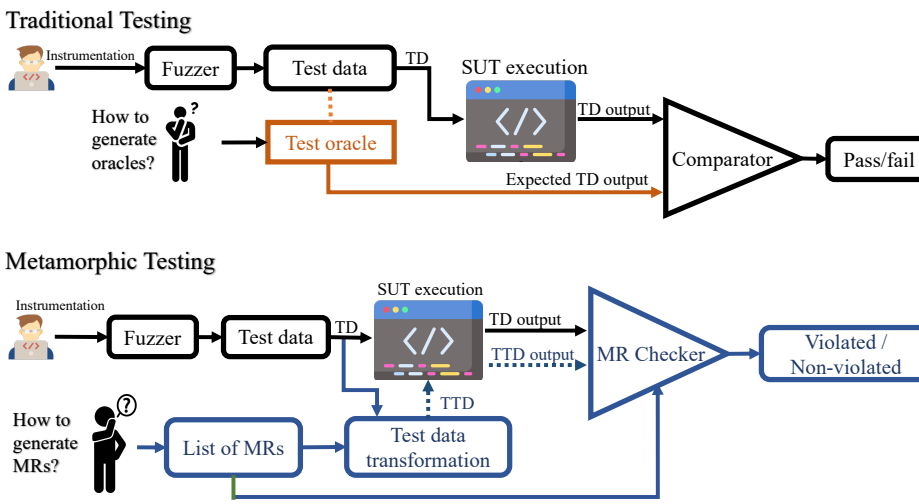

Figure 1. High-level visual workflow comparison: traditional testing vs metamorphic testing — from generating initial TD to checking result validity

2.1. Problem Statement and Research Goals

Generating *"good"* MRs is not trivial. Various approaches have been proposed for the generation of MRs. They can be divided into two groups: approaches that focus on the semantic correctness of the MR, and approaches that focus on its syntactic correctness (see Figure 2). Generating semantically correct MRs involves a deeper understanding and control of the SUT's behaviour, which re-

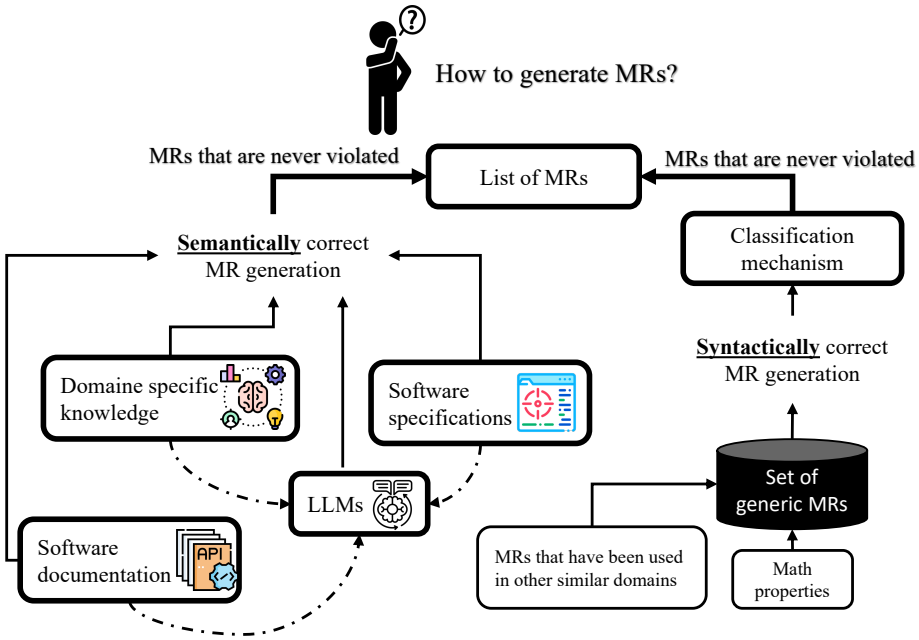

Figure 2. Approaches for Generating MRs: semantically correct generation, using domain-specific knowledge, software specifications, software documentation, and interactions with LLMs; and syntactically correct generation, utilising classification mechanisms and generic MRs. Both aim to produce MRs that are never violated.

quires broad domain-specific knowledge. The approaches belonging to this group leverage domain-specific knowledge such as software requirements specifications [23]—including functional and non-functional requirements—and software documentation [24], [25]—such as user manuals, API documentation, and technical specifications. Recently, Large Language Models (LLMs) have also been utilised for generating MRs, either by querying the LLM for MRs based on its built-in knowledge and SUT descriptions [26], [27], or by providing SUT specification documents to the LLM to derive MRs [28]. A semantic approach is predominantly performed manually, relying on the expertise and knowledge of the testers or developers involved.

On the other hand, approaches that focus on generating syntactically correct MRs centre around matching syntactic markers such as input types and code structures of the SUT with predefined or generic MRs. By "generic MRs", we refer to MRs that, although originally formulated for different contexts, share similarities in terms of input types or internal behaviours, allowing them to be indirectly applicable to various SUTs regardless their application domain. This group employs classification mechanisms to ensure that the generated MRs adhere to the syntactic characteristics of the SUT. Generating MRs based on syntactical correctness enables efficient reuse of MRs and simplifies the overall MT process. It

is important to note that both semantically and syntactically focused generation approaches aim to create MRs that never encounter violations for any valid MRs.

An early pioneering work demonstrating the feasibility of generating MRs through syntactically-correctness-based approaches was conducted by Kanewala *et al.* [30]. They introduced Predicting Metamorphic Relations (PMR), an approach that utilises ML techniques to classify whether a specific MR from a set of MRs is applicable (*i.e.,* never violated) to a given SUT. The core idea behind PMR is to develop a model that predicts whether a method in a newly developed SUT can be effectively tested using a specific MR, based on the method's CFG. Several subsequent works have followed the PMR approach; for instance, Hardin *et al.* [31] extended the initial PMR study using semi-supervised learning techniques on a set of CFG-based features tagged with six predefined MRs. Rahman *et al.* [22] applied the PMR approach for predicting three pre-defined MRs for matrix-based programs. Zhang *et al.* [32] introduced RBF-MLMR, a multi-label method for predicting MRs using radial basis function neural networks. Unlike PMR, which employs several binary classifiers, RBF-MLMR predicts all possible MRs for a given method. While RBF-MLMR uses a different approach than PMR, it follows the same PMR methodology and the same feature extraction approach.

While the aforementioned approaches have demonstrated promising results, they come with notable limitations:

1. *Dependency on Labelled Datasets:* Many of these approaches rely on binary classifiers, which require labelled datasets for effective learning. Creating such labelled datasets can be a time-intensive and resource-demanding process, making it impractical in scenarios where obtaining labelled data is challenging or costly.
2. *Limitation in Feature Extraction:* Another limitation is encountered during the feature extraction process for model training, especially when relying on CFG or source code metrics. These methods often overlook the possibility of code refactoring. This oversight is significant because code refactoring can alter the structure of the code, potentially making previous ML models irrelevant or less accurate.
3. *Assumption of Universal Applicability:* The logic behind the selection process using binary classifiers assumes that a chosen MR must universally apply to the entire valid input data space. This assumption may not always hold true. An MR that applies to specific input data might not apply to others within the same valid input data space. Relying on the belief that an MR must consistently apply across the entire input data space can lead to *false positives*, where an MR violation incorrectly suggests a fault when there is none.

In addition to the previous limitations, current practice in MT involves interpreting their outcomes, *i.e.,* whether the MR is violated or not, is largely a manual effort. This manual interpretation can be time-consuming and resource-intensive.

Furthermore, the cost of MT is directly influenced by the number of MRs used. As the number of MRs increases, the number of test cases may grow exponentially. This leads to longer execution times and a greater need for manual inspection of MT outcomes [33], [34].

Driven by these challenges, the goal of this thesis is to support the syntactic-based generation of MRs, refinement of MRs by applying constraints based on the TD, and assess the strengths of the generated and refined MRs.

2.2. Research Approach

We tackle our overall goal by addressing the following Research Goals (RGs):

- **RG_1 - MR Classification:** To provide a method for classifying MRs based on TD.
- **RG_2 - MR Constraint Definition:** To provide a method for refining MRs by setting constraints based on TD.
- **RG_3 - MR Strength Assessment:** To provide a method for assessing the strengths of the selected and refined MRs.

In the following subsections, we delve deeper into each RG, and introduce the Research Questions (RQs), and provide an overview of the work conducted in this thesis.

2.2.1. RG_1 - MR Classification

To effectively address RG_1, it is crucial to establish a baseline approach first. This baseline approach will serve as a benchmark against which our proposed method can be evaluated. Furthermore, it will help us pinpoint potential areas for enhancement in our approach. Therefore, we chose the pioneering work by Kanewala *et al.* [30], which demonstrated the feasibility of generating MRs through syntactically-correctness-based approaches, specifically the PMR method. This selection of PMR as our baseline allows us to build upon existing knowledge and methodologies in the field of MR generation. It provides a solid foundation from which we can develop and refine our own proposed method.

As discussed in Section 2.1, the idea behind PMR is to develop a model capable of predicting whether a specific MR can be used to test a method in a newly developed SUT. To evaluate the generalisability of PMR across various programming languages, we conducted a replication study on PMR [35]. Our replication study involved reconstructing the preprocessing and training pipeline. The results of our replication study validated the reported findings and laid the groundwork for subsequent experiments. In addition to this, we explored the potential reusability of the PMR model initially trained on Java methods. We assessed its suitability for functionally identical methods implemented in Python and C++. While the PMR model demonstrated strong performance with Java methods, its prediction

accuracy notably declined when applied to Python and C++ methods. Nevertheless, we found that retraining the classifiers using CFGs specific to Python and C++ methods led to improved performance. Furthermore, we conducted an evaluation of the PMR approach, considering source code metrics as an alternative to CFG for building the models [7]. Our results also led to the conclusion that a generalisation of PMR beyond unit testing, such as its application to system-level testing, does not appear to be feasible.

Building upon the findings and lessons learned from previous work, we explored the possibility of classifying MRs based on TD and the information gathered during the MT process itself. This involves the transformation of TD, the execution of TD and Transformed Test Data (TTD), and the comparison of their corresponding outputs. We then analyse the outcomes of this process and investigate all potential factors influencing the violation/non-violation outcomes. We initially validated this concept in [36] using a toy example. This evaluation confirmed that MRs may not universally apply to the entire TD input space. An MR violation may occur not because there is a defect in the SUT, but due to a discrepancy between the TD and the MR itself. Therefore, we introduced a new concept called *mixed cases*. These are cases in which the applicability of MRs is not clear-cut or straightforward.

We expanded on these initial findings by proposing and formalising a TD-driven method for classifying MRs. Similar to PMR, we assume the existence of a predefined list of MRs. However, our TD-driven method does not rely on labelled datasets and acknowledges that an MR may only be applicable to TD with specific characteristics. We have named our method MetaTrimmer.

MetaTrimmer consists of three steps: (1) TD Generation, which is responsible for generating long sequences of TD for the SUT. (2) MT Process, which is in charge of performing necessary TD transformations based on the predefined MRs. It generates logs to record information about TD inputs and its respective outputs, and any MR violations during the execution of the TD and the TTD against the SUT. (3) MR Analysis, which involves manual inspection of violation and non-violation results. In MetaTrimmer, an MR violated 100% of the time is considered as not applicable to the SUT, while an MR not violated in 100% of cases is applicable to the SUT. The mixed cases arise when MR violations and non-violations do not reach 100%. We showed how TD can be used in the selection of MRs, and by comparing it with the baseline approach, we demonstrated its effectiveness. Additionally, we answered the following research questions:

- **RQ$_{1.1}$:** How can TD be used to classify MRs?
- **RQ$_{1.2}$:** How well does the TD-driven MR classification method perform?

2.2.2. RG_2 - MR Constraint Definition

An important contribution of this research lies in challenging the notion that an MR must universally apply to the entire valid TD space. Instead, it highlights the relevance of MRs within specific subsets of this space. The rigid belief that an MR must encompass the entire input data space can limit its effectiveness. Real-world scenarios often feature distinct behaviours and characteristics across various subsets of the input data space. Recognising these subsets and introducing constraints based on test input data serves as a means to enhance the effectiveness of MRs. Additionally, emphasising the provision of explanations for MR violations emerges as a crucial aspect. It plays a pivotal role in distinguishing whether a violation stems from a code failure or an MR-imposed constraint. This process involves the identification of patterns, trends, and underlying causes of MRs violation.

In Section 2.2.1, related to RG_1, we have introduced MetaTrimmer, a TD-driven approach for classifying MRs. MetaTrimmer operates without needing labelled datasets and acknowledges that MRs may not always apply to the entire input data space but instead to specific subsets, known as constraints. In MetaTrimmer, an MR violated 100% of the time is considered as not applicable to the SUT, while an MR not violated in 100% of cases is applicable to the SUT. Scenarios where violations and non-violations do not reach 100% are referred to as mixed cases. These mixed cases, being less straightforward, provide valuable insights into the behaviour of MRs. Therefore, by analysing them can aid in generating constraints by revealing patterns and exceptions in the behaviour of the SUT relative to specific MRs.

As previously mentioned, MetaTrimmer consists of three main steps: (1) TD Generation, (2) MT Process, and (3) MR Analysis. RG_2 focuses specifically on the MR Analysis step. In this step, the analysis involves manually examining the outcomes of MR violations and non-violations to understand the reasons behind the MR's violation status. This examination aims to identify specific relevant TD or ranges, which aids in deriving constraints. While manual inspection provides valuable insights during MR Analysis, it may not always yield a completely accurate or comprehensive understanding of why MRs exhibit certain violation statuses. This limitation becomes more pronounced when dealing with large volumes of TD or multiple MRs. In such cases, there's a risk of extracting incomplete or incorrect constraints, which can impact the effectiveness of subsequent testing efforts. Furthermore, it's important to emphasise that discerning whether an MR violation is attributed to a fault in the SUT or due to the MR's inability to meet specific behavioural expectations of the SUT for particular TD is not unique to MetaTrimmer alone. This challenge is inherent in the broader context of MT technique.

Building upon the MetaTrimmer process, we introduced an approach employing Association Rule Mining (ARM) [37] to support the pattern extraction pro-

cess, thereby enhancing the MR Analysis and reducing the need for manual inspection for defining constraints. We demonstrated how TD, combined with ARM, can provide insights in the form of rules, enabling the definition of precise constraints that delineate the input space where each MR is valid. This process improves the accuracy and applicability of MRs while reducing dependency on extensive manual analysis. We answered the following research question:

- **RQ$_{2.1}$:** How can patterns be extracted and utilised as constraints to refine MRs?
- **RQ$_{2.2}$:** How well does the ARM-driven approach refine and constrain MRs?

2.2.3. RG$_3$ - MR Strength Assessment

The goal of MetaTrimmer is to produce a refined final set of MRs. Once this set is established, assessing its effectiveness becomes a natural next step. Mutation testing has been widely recognised as a robust method for evaluating the quality of test suites across various software testing levels (unit, integration, and specification) and for different programming languages. It operates on the principle of introducing artificial faults (mutants) into the codebase and then assessing whether the existing test suite can detect these faults. Mutation testing defines a quality metric known as the mutation score, ideally aiming for all mutants to be "killed" by the test suite (*i.e.,* achieving a mutation score of 1).

Pioneering work by Asrafi *et al.* [38] has provided a theoretical evaluation of MR effectiveness, primarily through mutation analysis and code coverage metrics. In another approach, Jia *et al.* [39] proposed a framework to assess the effectiveness of MT in the context of image processing. They used real image datasets from published libraries to generate test inputs. They applied mutation testing to evaluate fault detection rates through mutation score, demonstrating MT's ability to detect faults in edge detection programs with up to 90% accuracy. Jafari *et al.* [40] evaluated the fault detection MRs using mutation testing following Jia *et al.* [39] framework.

It is evident that mutation testing is widely adopted for assessing effectiveness in software testing. When evaluating the effectiveness of MRs, the traditional focus has centred on metrics such as mutation score and code coverage. However, these metrics may not always yield actionable insights across diverse application domains or testing levels. Therefore, we propose a three-level effectiveness assessment strategy by integrating MetaTrimmer with mutation testing.

- Level one focuses on evaluating MRs based on their ability to detect equivalent mutants.
- Level two examines MRs based on their capability to trigger violations when mutants are killed across the entire input space.
- Level three focuses on the sensitivity of MRs by analysing the amount of TD required to trigger a violation, highlighting the precision and adaptability of MRs to diverse input space.

We validated our assessment strategy using a SUT employed in an industrial setting, which underscores the relevance of our findings to real-world applications. In addition, we answered the following research question:

- **RQ$_{3.1}$:** What TD-based method can be used to assess the strengths of generated and refined MRs regarding their bug-finding capability?

2.3. Contribution of the Thesis

In this thesis, we present three main contributions consisting of an approach for classifying MRs based on TD, a method for refining MRs by setting constraints based on TD, and a three-level strategy for assessing the effectiveness of MRs.

Figure 3 illustrates the three contributions as dotted boxes. Contribution 1 covers the approach for classifying MRs based on TD. Contribution 2 focuses on the method for refining MRs by setting constraints based on TD. Contribution 3 presents the three-level strategy for assessing the strengths of the generated and refined MRs. Each contribution is visually separated, highlighting their respective components and processes.

- **Contribution 1** A TD-driving approach, MetaTrimmer, for classifying MRs.
- **Contribution 2** An ARM-based approach for refining MRs by setting constrains based on TD.
- **Contribution 3** A three-level strategy for assessing the strength of MRs

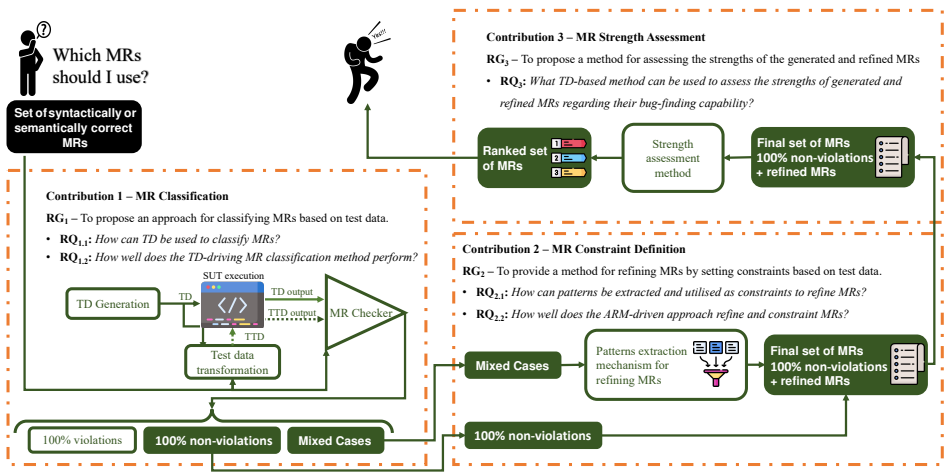

Figure 3. Research approach

The thesis presents three interrelated contributions: MetaTrimmer for MR classification, MetaTrimmer+ for refining MRs, and an assessment framework for evaluating the strength of MRs based on fault-detection capabilities. MetaTrimmer serves as the foundation by classifying MRs from a broader set into three categories: 100% violation, 100% no violation, and mixed cases. MetaTrimmer+

then refines the MRs that present mixed cases to ensure they are fully applicable. Finally, the assessment framework evaluates the fault-detection strength of the 100% no violation MRs and the refined MRs. This integrated process produces a final ranked set of MRs.

2.4. Structure of the Thesis

The introduction chapter has provided the context for this thesis, outlined the RGs and the RQs, presented our research approach, and has described contribution of the thesis. In Chapter 3, we introduce the key concepts relevant to this thesis, including MRs, TD generation, mutation testing, and ARM, which form the foundation for the proposed methods and approaches . Chapter 4 summarises the related work in the field, positioning our research within the area of MT. It particularly focuses on the selection and classification of MRs, reviewing prior contributions and identifying gaps that this thesis aims to address.

In Chapter 5, we present our first contribution, which introduces MetaTrimmer, a TD-driven approach for classifying MRs. The chapter details how TD can be used to dynamically classify MRs based on their behaviour across different input sets. Chapter 6 present our second contribution, which builds on MetaTrimmer by refining MRs through the introduction of constraints based on TD characteristics. The chapter explains how ARM is used to extract patterns and define these constraints. In Chapter 7, we present the third contribution, which focuses on assessing the strength of MRs. The chapter details a three-level strategy for evaluating MR effectiveness, integrating mutation testing to rank MRs according to their fault-detection capabilities. In addition, we detail the industrial scenario in which this strategy was applied, demonstrating its practical relevance in a real-world context.

Chapter 8 provides a discussion of the key findings, offering insights into the implications of the proposed methods, their limitations, and potential applications in real-world scenarios. In Chapter 9, we conclude the thesis by summarizing the contributions and discussing avenues for future research.

3. BACKGROUND

This chapter we introduce the key concepts necessary in our study. Section 3.1 introduces the MT approach. Section 3.3 provides a brief description of mutation testing and Section 3.4 gives a brief description of ARM. Section 3.5 provides the details description on the PMR approach.

3.1. Metamorphic Testing

Metamorphic Testing (MT) is a software testing technique introduced by Chen *et al.* [6] to address the test oracle problem. The idea behind MT is to explore the internal properties of the SUT to verify expected outputs or generate new test cases. The core of MT lies in exploring the relations between inputs and outputs across multiple executions of the SUT; such relations are known as MRs. Formally, a MR is a property that relates the input and output of the SUT to another set of input and output [7].

An MR consists of two parts, the *input transformation statement* and the *expected output change*. The *input transformation statement* of the MR specifies how to modify a given input to create a new related output which then will be compared to the unmodified, original input. The *expected output changes* defines the relation that the outputs must exhibit given the input modification. For example, let's consider a program that sorts a list L. To define an MR, one could state it as follows:

*"**IF** the list L is permuted to produce a new list L' **THEN** sorting L and L' should yield the same sorted list L''*. This can be expressed in mathematical notation as:

$$\forall L' \in P(L) : \text{sort}(L) = \text{sort}(L') = L'',$$

where $P(L)$ represents all possible permutations of L.

In other words, sorting any permutation L' of the list L, will always produce L''. If, in a given situation, the sorting program produces a different sorted list from a permutation of the initial input, it indicates a violation of the MR, which suggests a high probability of a fault in the SUT. However, it is essential to note that the lack of MR violations does not necessarily guarantees a fault-free SUT. Figure 4 illustrate the MT basic workflow involving five steps:

1. Generation of the initial TD.
2. Identification/generation of MRs that the SUT should satisfy.
3. Generation of the TTD by applying selected MR-specified transformations to the initial TD.
4. Execution of corresponding TD and TTD pairs.
5. Verification of whether the observed changes in the output during TD and TTD executions align with the changes defined by the applied MRs.

The final step requires further analysis to determine the outcome of the MT workflow, as no violations do not guarantee that the SUT is implemented correctly. If an MR is violated, it suggests a fault in the SUT, assuming the MR is correctly defined and is valid for the defined input data space [36].

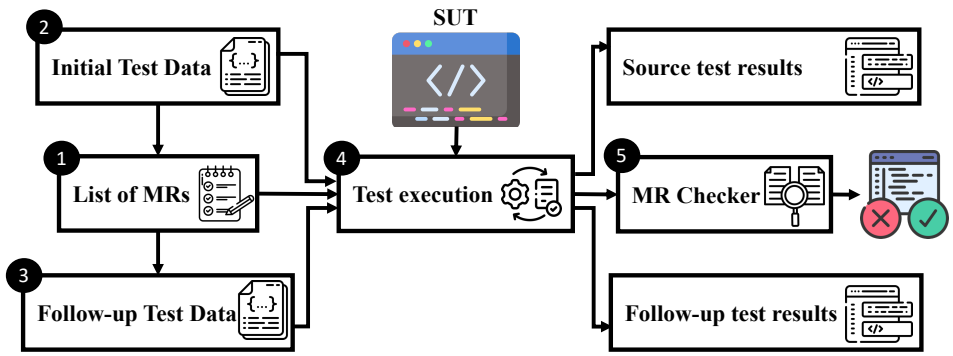

Figure 4. Metamorphic testing workflow.

3.2. TD Generation

In software testing, TD refers to the input data used during test execution. It is utilised for positive testing to verify that the functions of the SUT produce expected outputs for specific inputs and for negative testing to evaluate the SUT's ability to handle unusual or unexpected inputs. Insufficiently constructed TD could result in some potential test cases being missed, negatively impacting the quality of the software. This thesis does not aim to introduce new techniques for TD generation. Instead, we leverage fuzz testing for generating TD. Fuzz Testing is a software testing technique that injects erroneous or random data into software systems to discover coding errors and security vulnerabilities [41]. Fuzzing consists of three main components: the input generator, executor, and defect monitor. The input generator provides the executor with various inputs. The executor runs target programs on these inputs. The defects monitor the execution to determine whether it discovers new execution states or defects [41].

3.3. Mutation Testing

Since its introduction in the 1970s, mutation testing has been extensively studied and used reaching a maturity phase and gradually gains popularity both in academia and in industry [42]–[44]. Numerous studies have demonstrated that mutation testing is a robust method for identifying high-quality test suites across various software testing levels (unit, integration, and specification) and for different programming languages [45]. Moreover, it has been shown to be effective in

guiding test generation [46], [47], simulating real faults in software testing experimentation [48], [49], and localising faults [50], [51]. Overall, mutation testing serves as a fault-based testing technique that allows for the definition of a test suite quality metric, known as the mutation score. Typically, mutation testing involves three main steps:

1. *Mutant Generation:* From a correct program or SUT, several faulty versions, the so-called *mutants*, are automatically generated by introducing small changes into the code of the original SUT. There exist different ways of creating mutants, *e.g.,* changing variable names, changing operands and operators, removing or adding code lines, and so on.
2. *Mutant Execution:* To check whether the test suite can detect the mutants generated, the test suite is run on each mutant. If a test in the test suite fails, the mutant has been *killed*, if no test fails, the mutant *survived*.
3. *Mutation Score Calculation:* The mutation score is calculated by dividing the amount of killed mutants by the total amount of mutants.

Ideally all mutants are killed by the existing test suite, *i.e.,* the mutation score equals 1. Numerous tools for generating mutants have been developed for various programming languages [44]. Examples include *MuJava* [52], *Major* [53] and *PIT* [54] for Java, *Mutatest* [55] and *MutPy* [56] for Python, and *Milu* [39] for the C language. However, for MATLAB programs, as far as we know, there is just one tool. MATmute is an automatic code mutator for MATLAB, created by Hook *et al.* [57] to support their research in mutation sensitivity testing in 2009. Despite its initial development, the tool has not seen updates since 2015. For MATLAB Simulink, more recent tools like MUT4SLX [58] and *SIMUTATOR* [59] are available. Typically, these tools offer various types of mutation operators (predefined rules for making changes to the original SUT to create mutants), support for reducing the number of mutants and avoiding equivalent mutants, *i.e.,* mutants that are syntactically different from the original SUT but show identical behaviour, and optimisations for speeding up the mutant generation and test process [45].

The high computational costs are considered as one of the main factors hindering the wider adoption of mutation testing within industry [60]. In the mutant execution step, mutation testing requires running the tests from the test suite against each of the generated mutants. In real-world projects, a large number of mutants can be generated. For example, as reported in [61], mutation testing applied for a safety-critical embedded system with 60 KLOC produced 75043 mutants. Thus, the maximum possible number of test runs is usually huge. It is equal to the number of tests times the number of mutants (full kill matrix). Even though that in practice the number of test runs can be significantly reduced when tests are executed until one of them eventually kills the mutant, the remaining number of test executions is usually still very high. This is especially true, if there is a large number of surviving mutants, as for each of them all the tests of the test suite have to be run.

3.4. Association Rule Mining

Association Rule Mining (ARM) is a rule-based unsupervised ML method that enables the discovery of relations between variables or items in large databases [37]. ARM has found applications in various fields, including business analysis, medical diagnosis, and census data analysis, to uncover previously unknown patterns. ARM process typically involves two major steps: finding all frequent itemsets that satisfy minimum support thresholds and generating strong association rules from these frequent itemsets by applying minimum confidence thresholds. Bellow, we define important terminology:

- ***Itemset:*** An itemset, denoted as $I = \{X, \ldots, Y, Z\}$, represents a set of different items in the dataset D, typically consisting of k distinct items.
- ***Association rule:*** An association rule reveals the relationship between items in a dataset D containing n transactions with sets of items.
- ***Support:*** The support indicates the percentage of transactions in dataset D containing both itemsets X and Y. For an association rule $X \rightarrow Y$, the support is calculated as:

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

- ***Confidence:*** Confidence represents the percentage of transactions in the database D with itemset X that also contains itemset Y. It is calculated using conditional probability, expressed in terms of itemset support:

$$confidence(X \rightarrow Y) = P(Y|X) = support(X \cup Y)/support(X)$$

- ***Lift:*** Lift measures the frequency of occurrence of X and Y together, considering their statistical independence. The lift of rule $(X \rightarrow Y)$ is defined as $lift(X \rightarrow Y) = confidence(X \rightarrow Y)/support(Y)$. A lift value of one indicates that X and Y appear together as frequently as expected under the assumption of conditional independence.
- ***Zhang Metric:*** The Zhang metric, introduced in by Xiaowei Yan *et al.* [37], provides a comprehensive measure of the quality and significance of association rules in ARM. Defined as:

$$zhangs\_metric(A \rightarrow C) = \frac{conf(A \rightarrow C) - conf(A' \rightarrow C)}{\max[conf(A \rightarrow C), conf(A' \rightarrow C)]}$$

  The Zhang metric measures both association and dissociation between items A and C. With a range of $[-1, 1]$, a positive value (> 0) indicates association, while a negative value indicates dissociation.

3.5. Predicting Metamorphic Relations

In this section, we first present the Predicting Metamorphic Relations (PMR) procedure proposed by Kanewala *et al.* [29](Section 3.5.1). Then, we present a detailed description of the set of pre-defined MRs used in the original and our study

(Section 3.5.2) as well as the labelled dataset used in the original study (Section 3.5.3). Finally, we summarise the evaluation results reported in the original study (Section 3.5.4).

3.5.1. The PMR Procedure

The goal of the PMR approach is to build a model that predicts whether a method in a newly developed SUT can be automatically tested by exploiting one or more MRs contained in a pre-defined set of MRs. Figure 5 shows the PMR procedure. The PMR procedure consists of three phases. *Phase I* is responsible for creating a graph description representation derived from a method's CFG. The output of this phase is a *DOT* file. *Phase II* is in charge of feature extraction from the method's DOT file. Also, each method is labelled with elements from the set of pre-defined MRs. Thus, the output of this phase is a labelled dataset. *Phase III* is in charge of training and evaluating the binary classification models that predict whether a specific MR is applicable to the unit testing of a specific method. Below we describe each phase in detail.

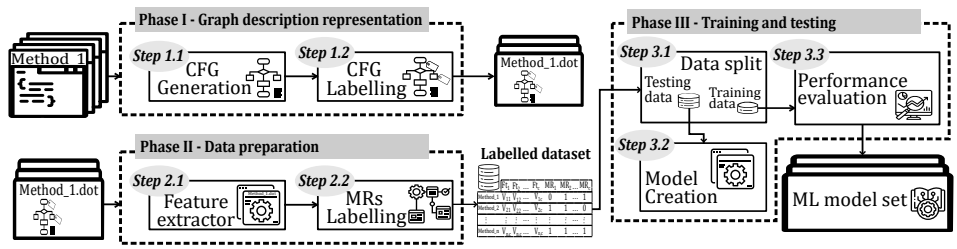

Figure 5. PMR procedure

Phase I – Graph description representation. This Phase starts with the creation of the graph representation from the method's source code. Then a labelled CFG is created by annotating each node in the CFG. The process can be split into the following two steps.

Step 1.1 – CFG generation: This step is responsible for creating the CFG from the method's source code. To get the CFG, Kanewala *et al.* use a java tool called *Soot* [62]. *Soot* generates CFG representations in *Jimple* format, a typed 3-address intermediate representation, where each CFG node represents an atomic operation [62]. The left-hand side of Figure 6 shows the CFG representation of the Algorithm 1 using the *soot* framework. The numbering of the nodes has been done manually, *i.e.,* not with the framework, and serves here only to facilitate a better understanding of the next phase and its steps.

Step 1.2 – CFG labelling: In this step a simplified version of the CFG is created by replacing the specific, code-related information of each node in the CFG by a more general annotation describing the specific operations and conditional jumps in the original code. The right-hand side of Figure 6 shows the CFG representation with the node annotations of the Algorithm 1. Table 1 shows examples

Algorithm 1 Average of an integer array

```
1: function AVG(int input[ ])
2:     double sum = 0;
3:     double average = 0;
4:     for (int i = 0; input.length; i++) do
5:         sum + = input[i];
6:     average = sum/input.length;
7:     return average
```

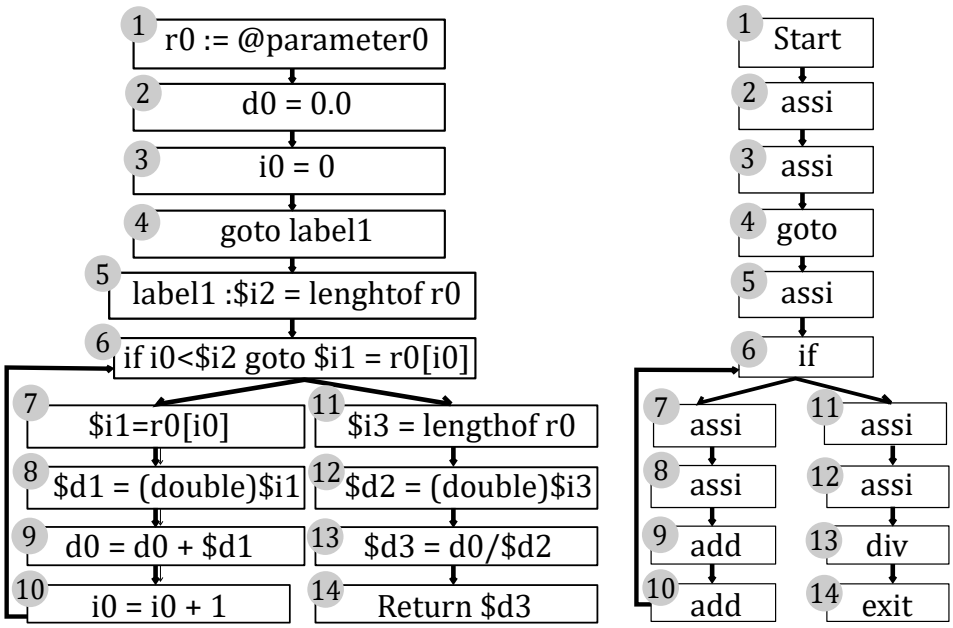

Figure 6. On the left, CFG representation of Algorithm 1 using the *soot* framework; on the right, its CFG with annotations

of annotations that are assigned depending on the node operation. The annotations follow the graph description language, *i.e.,* the DOT format.

Phase II – Data preparation. This phase is in charge of extracting a set of features from the annotated CFGs, *i.e,* from the Phase I output. Also, to each method's annotated CFG zero to six pre-defined MRs are assigned, depending on their suitability. Like Phase I, Phase II consists of two steps.

Step 2.1 – Feature extraction: Kanewala *et al.* propose two approaches for extracting features from CFG representations, features based on nodes and paths, and features based on graph similarity measures. In the former, the node features (hereafter simply NF) follows the form $NO_n - d_{in} - d_{out}$, where NO_n stands for Node Operation of node n, and d_{in} and d_{out} stand for *in-degree* and *out-degree*, respectively. The number of a specific NF type, *i.e.,* $NO_n - d_{in} - d_{out}$, is tallied and used as the NF value. As an example of NF, let us consider the annotated CFG

of Algorithm 1. As the right-hand side of Figure 6 shows, the annotated CFG of Algorithm 1 has fourteen nodes. Among these fourteen nodes, there are seven with the type annotation *assi*, two with type annotation *add*, and five with unique type annotations, *i.e, start, goto, if, div* and *exit*. For each node, the d_{in} and d_{out} are calculated, too, to derive the complete NF. For instance, the node *start* (node 1) will be represented by the NF *start-0-1*, where *start* is NO_1, 0 is d_{in} and 1 is d_{out}. Each unique NF is tallied to get the corresponding NF value. For *start-0-1*, the NF value is 1. Table 2 shows the set of NFs and their values extracted from Algorithm 1 using its CFG representation.

The feature based on path information (hereafter just PF) refers to the shortest routes from the start node to each node in the graph, as well as the shortest path from each node in the graph to the end node. This feature follows the form $NO_1 - NO_2 - NO_{...} - NO_n$, where NO_n, as in the NF, denotes a specific operation statement in node n. The value of each PF is the number of occurrences of each path in the CFG. For instance, let us consider the labelled CFG of Algorithm 1. As Figure 6 right side shows, the path composed by the nodes the PF *1-2-3-4-5-6-7* and the nodes *1-2-3-4-5-6-11* can be denoted as *start-assi-assi-goto-assi-if-assi*. Therefore, its feature value is 2 since there are two paths represented by one type of PF. Table 3 shows the set of PFs and their associated PF values extracted from Algorithm 1 using its CFG representation.

The second approach to extract features from CFG is by using *graph similarity measures*. Graph similarity refers to the process of determining the degree of similarity between two or more graphs. In particular, Kanewala *et al.* use Random Walk Kernel (RWK) and Graphlet Kernel (GK). RWK is the most-studied family of graph kernels [63]. It provides measure the similarity between two or more graphs based on the number of common walks in the graphs. The concept behind GK is to randomly sample tiny (connected) sub-graphs of size k, and using them to compare frequency distributions or to construct graph invariants.

Table 1. Node operations (NO) in the control flow graph and corresponding labels (CL) for the annotation

NO	CL	NO	CL	NO	CL	NO	CL
+	add	−	sub	∗	mul	/	div
||, *or*	or	&, *and*	and	*if*	if	=	assi
==	eql	>=	geql	>	gt	<=	leql
<	lt	!=	neql	:=	start	%	rem
invoke	fcall	*return*	return	*exit*	exit	*goto*	goto

Step 2.2 – MR labelling: The key idea of PMR is predicting whether a given method is suited for a particular MR by using binary classifiers. PMR uses supervised learning classification algorithms, *i.e.,* a labelled dataset is needed to provide examples for learning. Thus, after *Step 2.1 – Feature extraction*, the training dataset is created by manually labelling each method with applicable MRs. De-

pending on whether a specific MR does or does not satisfy the method, the method is labelled with 1 or 0 for this MR, respectively.

Phase III – Training and testing. This phase involves the use of one or more supervised machine learning (ML) algorithms, or a combination of them, to derive knowledge from the data. Three steps needs to be conducted.

Step 3.1 – Data split: This step is responsible for splitting the dataset into two subsets: a training set and a test set. The training set is used to create the prediction model, while the test set is used to evaluate the performance of the created prediction model.

Step 3.2 – Model creation refers to the process of building prediction models. Choosing a good modelling technique is vital for the training and prediction stage in any ML application, including the PMR approach. Kanewala *et al.* get the best results using the Support Vector Machine (SVM) technique.

Step 3.3 – Performance evaluation: This step measures the performance of the created prediction models. Performance measures are derived from the *Confusion Matrix*. Let A denote a classification output in which a specific MR_n satisfies the method m, and let A' denote a classification output in which a specific MR_n does not satisfy the method m, then A can be seen as the *positive class* and A' as the *negative class*. Using this notation, each standard performance measure is expressed as a function of the counts of elements in the *Confusion Matrix* defined as follows:

- ***True Positive (TP):*** The actual MR of a method was A and the predicted was A. This represents a successful prediction.
- ***True Negative (TN):*** The actual MR of a method was A', the predicted was A'. This represents a successful prediction.
- ***False Positive (FP):*** The actual MR was A' and the predicted was A. This represents an unsuccessful prediction.
- ***False Negative (FN):*** The actual MR was A and the predicted was A'. This represents an unsuccessful prediction.

Accuracy is the ratio of successful predictions made to both classes and expressed as:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.1}$$

Precision (or positive predictive value) is the ratio of correct predictions made for class A and is shown in Equation (3.2):

Table 2. Node features extracted from Algorithm 1 related to the nodes of its CFG representation

Node Feature	Node Feature Value	Node Feature	Node Feature Value
start-0-1	1	*if-2-2*	1
assi-1-1	7	*add-1-1*	2
goto-1-1	1	*div-1-1*	1

Table 3. Path Features extracted from Algorithm 1 related to the paths of its CFG representation

Path Feature	Path Feature value
Shortest path from the start node to each node	
start	1
start-assi	1
start-assi-assi	1
start-assi-assi-goto	1
start-assi-assi-goto-assi	1
start-assi-assi-goto-assi-if	1
start-assi-assi-goto-assi-if-assi	2
start-assi-assi-goto-assi-if-assi-assi	2
start-assi-assi-goto-assi-if-assi-assi-add	1
start-assi-assi-goto-assi-if-assi-assi-div	1
start-assi-assi-goto-assi-if-assi-assi-add-add	1
start-assi-assi-goto-assi-if-assi-assi-div-exit	1
Shortest path from each node to the end node	
assi-assi-goto-assi-if-assi-assi-div-exit	1
assi-goto-assi-if-assi-assi-div-exit	1
goto-assi-if-assi-assi-div-exit	1
assi-if-assi-assi-div-exit	1
if-assi-assi-div-exit	1
assi-assi-div-exit	1
assi-div-exit	1
div-exit	1
exit	1
assi-assi-add-add-if-assi-assi-div-exit	1
assi-add-add-if-assi-assi-div-exit	1
add-add-if-assi-assi-div-exit	1
add-if-assi-assi-div-exit	1

$$Precision = \frac{TP}{TP+FP} \tag{3.2}$$

Recall (or true positive rate, or sensitivity) is the ratio of successful predictions made to cases of class A

$$Recall = \frac{TP}{TP+FN} \tag{3.3}$$

The *f-measure* statistic (or F1 score) is the harmonic mean of precision and recall:

$$\text{f-measure} = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{3.4}$$

In addition to the aforementioned performance measures, the *Area Under Curve* (AUC) and the *Balanced Success Rate* (BSR) measures are also widely used. The AUC is the area under the curve that plots the False Positive Rate (FPR) against the True Positive Rate (TPR) at different points in $[0,1]$. In binary classification problems, the BSR measure is calculated as the average of recall obtained on each class [64].

3.5.2. MRs Used in the PMR Study

In the original study, Kanewala *et al.* use six MRs that had been suggested previously in other studies [22], [30], [31], [65], [66]. Below we describe each MR in detail.

$$Input_{TD} = X_i, \ ...,X_n \text{ where } X_i \geq 0,\ 0 \leq i \leq n$$

The outputs of the source and follow-up test cases are written as $Output_{TD}(X)$ and $Output_{TTD}(Y)$, respectively.

The MRs based on these inputs and outputs are:

- MR_1: ***"Addition"*** (**ADD**). To get the ttd input, add a positive constant "C" to each element of the td input, *i.e.*,

  $$Input_{TTD} = X_1 + C,\ X_2 + C,\ X_3 + C..., X_n + C,$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) \geq Output_{TD}(X)$$

- MR_2: ***"Multiplication"*** (**MUL**). To get the ttd input, multiply each td input element with a positive constant "C", *i.e.*,

  $$Input_{TTD} = X_1 * C,\ X_2 * C,\ ...,\ X_n * C$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) \geq Output_{TD}(X)$$

- MR_3:***"Permutation"*** (**PER**). To get the ttd input, randomly permute the td input elements, *e.g.*, like

  $$Input_{TTD} = X_3,\ X_1,\ X_n,\ ...,\ X_2$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) = Output_{TD}(X)$$

- MR_4: ***"Inclusive"*** (**INC**). To get the ttd input, include a new element "$X_{n+1} \geq 0$" to the td input, *e.g.*, like

  $$Input_{TTD} = X_1,\ X_2,\ X_3,\ ...,\ X_n,\ X_{n+1}$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) \geq Output_{TD}(X)$$

- MR_5: ***"Exclusive"*** (**EXC**). To get the ttd input, remove an element "$X_{n-1} \geq 0$" from the td input, *e.g.*, like

  $$Input_{TTD} = X_1,\ X_2,\ X_3,\ ...,\ X_{n-1}$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) \leq Output_{TD}(X)$$

- MR_6: ***"Invertive"*** (**INV**). To get the ttd input, take the inverse of each td input element $X_i > 0$, *i.e.*,

  $$Input_{TTD} = 1/X_1,\ 1/X_2, 1/X_3,\ ..., 1/X_n$$

  Then the following output-relation must hold:

  $$Output_{TTD}(Y) \leq Output_{TD}(X)$$

3.5.3. Dataset Used in the PMR Study

In their original study, Kanewala *et al.* relied on a code corpus containing 100 Java methods that take numerical inputs and produce numerical outputs. The methods are from the open-source libraries *Colt Project* [67], which is an open-source library written for high-performance scientific and technical computing, *Apache Mahou* [68], which is a machine learning library, *Apache Commons Mathematics* [69], which is a Library of mathematics and statistics components, and *Java Collections* [70], which is a framework that provides an architecture to store and manipulate the group of objects. All of these libraries are written in Java.

To create a training dataset, Kanewala *et al.* manually labelled each method with the set of pre-defined MRs in a binary manner, *i.e.,* if MR_n matches a method m, then this method is assigned the label 1 for MR_n, otherwise it is 0.

Table 12 reports the total number of methods that do and do not match a specific MR. One sees that more than half of the methods match with MRs denoted as ADD, MUL and INV, while approximately one third of the methods match with MRs denoted as PER, INC, and EXC.

Table 5 reports how many methods have 0, 1, 2, ... 6 matching MRs and how those MRs are distributed in each case. 20 out of 100 methods have no matching MR, and only 9 out of 100 methods match with all six MRs simultaneously. Table 6 shows the names of all methods used in the study and the library to which they belong. It also shows whether or not a specific MR matches the method. For better readability, we use the symbol ✓ to denote that a specific MR matches, otherwise, we use the symbol ✗. In total, 26 methods stem from the *Colt project*, 8 are from *Apache Mahout*, 25 are from *Apache Commons Mathematics*, and 41

methods are from *Java Collections*. These methods are provided by Kanewala *et al.*[1] in the form of CFG representation, using the graph description language format DOT, instead of source code.

Table 4. Total number of methods that match (✓) and do not match (✗) a specific MR

MR	Change in the input	Output expected	✓	✗
ADD	Add a positive constant	Increase or remain constant	56	44
MUL	Multiply by a positive constant	Increase or remain constant	66	34
PER	Permute the components	Remain constant	33	67
INC	Add a new element	Increase or remain constant	34	66
EXC	Remove an element	Decrease or remain constant	32	68
INV	Take the inverse of each element	Decrease or remain constant	63	37

Table 5. Total number of methods that have 0, 1, 2, ..., 6 matching MRs and their distributions

No. MR†	No. Met⊥	ADD	MUL	PER	INC	EXC	INV
0	20	0	0	0	0	0	0
1	8	2	3	0	2	0	1
2	7	3	4	2	1	1	3
3	23	19	17	5	5	5	18
4	26	16	26	16	10	10	26
5	7	7	7	1	7	7	6
6	9	9	9	9	9	9	9

†Number of MRs that may apply to certain method, ⊥Number of methods

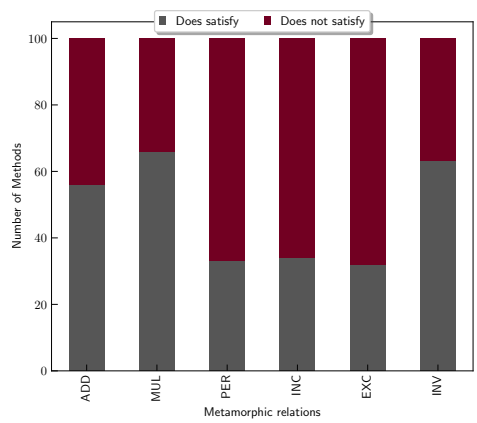

Figure 7. Total number of methods that satisfies or non-satisfies each MR

[1] http://www.cs.colostate.edu/saxs/MRpred/functions.tar.gz

Table 6. Labelled dataset [29]: symbol ✓denotes an MR-method match; symbol ✗ denotes that there is no match

ID	Method Name	Lib	Metamorphic Relation						ID	Method Name	Lib	Metamorphic Relation					
			ADD	MUL	PER	INC	EXC	INV				ADD	MUL	PER	INC	EXC	INV
1	add_values	Col	✓	✓	✓	✓	✓	✓	51	find_median	Col	✓	✓	✓	✗	✗	✓
2	array_calc	Col	✓	✓	✗	✗	✗	✓	52	find_min	Col	✓	✓	✓	✗	✗	✓
3	array_copy	Col	✓	✓	✗	✗	✗	✓	53	g_Test	Mth	✗	✓	✗	✓	✓	✓
4	autoCorrelation	Ct	✗	✗	✗	✗	✗	✗	54	geometric_mean	Col	✓	✓	✓	✗	✗	✓
5	average	Col	✓	✓	✓	✗	✗	✓	55	get_array_value	Col	✓	✓	✗	✓	✓	✓
6	bi_SearchFromTo	Ct	✗	✗	✗	✓	✗	✗	56	hamming_dist	Col	✗	✗	✗	✓	✓	✓
7	bubble	Mth	✓	✓	✓	✗	✗	✓	57	harmonicMean	Ct	✓	✓	✓	✗	✗	✓
8	cal_AbsoluteDiff	Mth	✓	✓	✗	✗	✗	✓	58	insertion_sort	Col	✓	✓	✓	✗	✗	✓
9	cal_Diff	Mth	✗	✗	✗	✗	✗	✗	59	kurtosis	Ct	✓	✓	✓	✗	✗	✗
10	chebyshevDist	Mh	✗	✓	✗	✓	✓	✓	60	lag	Ct	✗	✓	✗	✗	✗	✗
11	checkNonNegative	Mth	✗	✓	✓	✓	✗	✓	61	manhattanDist	Mh	✗	✓	✗	✓	✓	✓
12	checkPositive	Mth	✗	✓	✓	✓	✗	✓	62	manhattanDist2	Col	✗	✓	✗	✓	✓	✓
13	check_equal	Col	✗	✗	✗	✗	✗	✗	63	max	Ct	✓	✓	✓	✓	✓	✓
14	check_eq_tolerance	Col	✗	✗	✗	✗	✗	✗	64	meanDeviation	Ct	✓	✗	✓	✗	✗	✗
15	chiSquare	Mth	✗	✓	✗	✗	✗	✓	65	mean_Diff	Mth	✗	✗	✗	✗	✗	✗
16	clip	Col	✗	✗	✗	✗	✗	✗	66	mean_abs_error	Col	✗	✓	✗	✗	✗	✓
17	cnt_zeroes	Col	✗	✗	✓	✓	✓	✗	67	min	Ct	✓	✓	✓	✗	✗	✓
18	canberraDist	Mth	✗	✗	✗	✓	✓	✓	68	partition	Mth	✗	✗	✗	✗	✗	✗
19	cal_DividedDiff	Mth	✓	✗	✗	✗	✗	✗	69	polevl	Ct	✓	✓	✗	✓	✓	✓
20	cosineDist	Mh	✗	✓	✗	✗	✗	✗	70	pooledMean	Ct	✓	✓	✓	✗	✗	✓
21	count_k	Col	✗	✗	✓	✓	✓	✗	71	pooledVariance	Ct	✓	✓	✓	✗	✗	✓
22	count_non_zeroes	Col	✓	✓	✓	✓	✓	✓	72	power	Ct	✗	✓	✗	✗	✗	✓
23	covariance	Ct	✓	✗	✗	✗	✗	✗	73	product	Ct	✓	✓	✓	✓	✓	✓
24	dec	Mh	✗	✗	✗	✗	✗	✗	74	quantile	Ct	✓	✓	✗	✗	✓	✓
25	dec_array	Col	✓	✓	✗	✗	✗	✓	75	reverse	Col	✓	✓	✗	✗	✗	✓
26	Dist	Mth	✗	✓	✗	✓	✓	✓	76	safeNorm	Col	✓	✓	✓	✓	✓	✓
27	DistInf	Mth	✗	✓	✗	✓	✓	✓	77	sampleKurtosis	Mth	✓	✗	✓	✗	✗	✓
28	dot_product	Col	✓	✓	✗	✓	✓	✓	78	sampleSkew	Ct	✓	✗	✓	✗	✗	✗
29	durbinWatson	Ct	✗	✓	✗	✗	✗	✗	79	sampleVariance	Ct	✓	✓	✓	✗	✗	✓
30	ebeAdd	Mth	✓	✓	✗	✗	✗	✓	80	sampleWeightedVar	Ct	✗	✗	✗	✗	✗	✓
31	ebeDivide	Mth	✗	✗	✗	✗	✗	✗	81	scale	Ct	✓	✓	✗	✗	✗	✓
32	ebeMultiply	Mth	✓	✓	✗	✗	✗	✓	82	s_add	Mh	✓	✗	✗	✓	✓	✗
33	ebeSubtract	Mth	✗	✗	✗	✗	✗	✗	83	selection_sort	Col	✓	✓	✓	✗	✗	✓
34	elemtWise_equal	Col	✗	✗	✗	✗	✗	✗	84	sequential_search	Col	✗	✗	✗	✓	✓	✗
35	elemtWise_max	Col	✓	✓	✗	✗	✗	✓	85	set_min_val	Col	✓	✓	✗	✗	✗	✓
36	elemtWise_min	Col	✓	✓	✗	✗	✗	✓	86	shell_sort	Col	✓	✓	✓	✗	✗	✓
37	elemtWise_not_eq	Col	✗	✗	✗	✗	✗	✗	87	skew	Col	✓	✓	✓	✗	✗	✗
38	entropy	Mth	✓	✓	✓	✓	✓	✗	88	square	Col	✓	✓	✗	✗	✗	✓
39	equals	Mth	✗	✗	✗	✗	✗	✗	89	standardize	Col	✓	✓	✗	✗	✗	✗
40	errorRate	Mh	✗	✗	✗	✗	✗	✗	90	sum	Mh	✓	✓	✓	✓	✓	✓
41	euc_Dist	Mth	✗	✓	✗	✓	✓	✓	91	sumOfLogarithms	Col	✓	✓	✓	✓	✓	✓
42	evaluateHoners	Mth	✓	✓	✗	✓	✓	✓	92	sum_Power_Deviat	Ct	✗	✗	✗	✗	✗	✗
43	eval_Internal	Mth	✗	✗	✗	✗	✗	✗	93	sum_labeled	Ct	✗	✗	✗	✗	✗	✗
44	evalNewton	Mth	✗	✗	✗	✓	✗	✗	94	tanimotoDist	Mh	✗	✗	✗	✗	✗	✗
45	evalWeightedProd	Mth	✓	✓	✗	✓	✓	✓	95	variance	Col	✓	✓	✓	✗	✗	✓
46	find_diff	Col	✗	✗	✗	✗	✗	✗	96	var_Difference	Ct	✓	✓	✗	✗	✗	✓
47	find_euc_Dist	Col	✗	✓	✗	✓	✓	✓	97	weightedMean	Ct	✓	✓	✗	✗	✗	✓
48	find_magnitude	Col	✓	✓	✓	✓	✓	✓	98	weightedRMS	Ct	✗	✗	✗	✗	✗	✗
49	find_max	Col	✓	✓	✓	✓	✓	✓	99	weighted_average	Col	✓	✓	✗	✗	✗	✓
50	find_max2	Col	✓	✓	✗	✓	✓	✓	100	winsorizedMean	Ct	✓	✓	✗	✗	✓	✓

Col: *Java Colction*, **Mh:** *Apache Mhut*, **Mth:** *Apache Commons Mthematics*

3.5.4. Performance Achieved in the PMR Study

Kanewala *et al.* use features based on nodes and paths, as well as features based on graph similarity (RWK and GK), to build 18 binary SVM models, *i.e.,* three models (each using different feature sets) per any of the six specific MRs. They use AUC and BSR to evaluate model performance and consider AUC > 0.80 to be a good classification performance. Among the eighteen trained SVM models, the most promising one was when RWK was used. The average performance for the six models using RWK was 0.87 in terms of AUC.

4. RELATED WORK

MT has been demonstrated to be an effective technique for testing in a variety of application domains, *e.g.,* autonomous driving [8], [9], cloud and networking systems [12], [13], bioinformatic software [14], [15], scientific software [21], [71]. In the context of testing OA using MT, there have been a limited number of studies. However, the existing research demonstrates the effectiveness of MT as a testing technique for OAs. For instance, Yoo [11] applied MT to stochastic OAs and evaluated its impact on different problem instances, *i.e.,* specific selection of values for the algorithm parameters. The study focused on the Next Release Problem (NRP) and showed that MT could be effective in testing OAs, even considering their stochastic nature.

The application of MT to industrial-like scenarios has been extensively demonstrated in academic research. These studies, while academic in origin, validate MT's utility in addressing real-world challenges in industrial settings. Examples include testing navigation systems like Google Maps [72], optimizing cyber-physical systems [73], and verifying industrial control systems [74]. In the case of Google Maps [72], MT identified bugs in route optimization algorithms, including inconsistent cost calculations and non-optimal paths under certain transformations. For cyber-physical systems [73], MT revealed performance instability caused by environmental variations and nonlinear scaling issues in manufacturing processes. Similarly, for industrial control systems [74], MT detected synchronization bugs and unsafe state transitions due to parameter inconsistencies.

It is well known that the efficacy of MT heavily relies on the specific MRs employed. Some research has been done on how to choose "good" MRs. Chen *et al.* [75] examined several MRs discovered for shortest path and critical path programs, attempting to determine MRs that are useful. Liu *et al.* [76] introduced the Composition of MRs (CMRs) technique for constructing new MRs by mixing multiple existing ones. Zhang *et al.* [77] proposed a method in which an algorithm searches for MRs expressed as linear or quadratic equations. Chen *et al.* [78] developed METRIC, a specification-based technique and related tool for identifying MRs based on the category-choice framework. They also expanded METRIC into METRIC+ by integrating the information acquired from the output domain [79].

Kanewala *et al.* [30], were the first to show that, for previously unseen methods, applicable MRs can be predicted using ML techniques. Their work showed that classification models created using a set of features extracted from CFGs and a set of predefined MRs are effective in predicting whether a method in a newly developed SUT can be tested using a specific MR taken from the pre-defined set. Then, they extend their first work [30] by conducting a feature analysis to identify the most effective CFG's related features for predicting MRs [29]. Their results showed that SVM models built with features based on CFG similarity measurements, in particular using RWK, perform better than SVM models using nodes- and paths-based features with linear kernel.

Hardin *et al.* [31] extended the initial PMR study [30] using semi-supervised learning techniques on a set of node-based features and the CFG path tagged with six predefined MRs. Rahman *et al.* [22] applied PMR approach for predicting three high-level categories of MRs (*i.e.,* Permutative, Additive, and Multiplicative) for matrix-based programs. Their results show that the RWK can effectively predict these MRs. Nair *et al.* [80] explored and compared equivalent and non-equivalent mutants as data augmentation technique to broaden the training set using PMR. Their augmentation approach was tested on the PMR original study dataset [30]. The study demonstrated that equivalent mutants are a valid data augmentation technique to improve the PMR detection rate. Zhang *et al.* [32] presented RBF-MLMR, a multi-label technique that predicts MRs using radial basis function neural networks. Instead of using several binary classifiers like in PMR, RBF-MLMR use a neural network to predict all potential MRs for a given method. The major difference between this technique and PMR is the usage of multi-label and neural networks, but it follows the same pipeline as PMR original study. Also, the RBF-MLMR's feature design is CFG's node- and path-based.

Rahman *et al.*[81] introduce *MRpredT* which is a text classification-based ML approach to predict MRs using software documentation. The idea behind their MRpredT approach is to build a model that predicts whether a method in a newly developed SUT can be tested using a specific MR. A total of 93 program's Javadocs, which handle matrix operations, were used for their study. Then, text feature extraction methods are applied to those pre-processed Javadocs to obtain the feature vectors. These feature vectors of the programs are then supplied into the SVM and Naive Bayes classification algorithms with their associated MR labels. The MR labels are identified manually for all the programs. A disadvantage of MRpredT approach is the need of having a pre-defined set of MRs. Blasi *et al.*[24], proposed an approach supporting unit testing at the method level called MeMo, to *"automatically derive metamorphic equivalence relations from natural language documentation, and use such metamorphic relations as oracles in automatically generated test cases"*. MeMo infers MRs by identifying sentences in the comments that describe equivalent behaviours between different methods of the same class. Then, the inferred MRs get automatically translated into executable assertions.

Regarding finding constraints of MRs, Castro-Cabrera *et al.* [82] proposed a method that generates MRs based on two concepts: identifying potential input constraints and utilising constraint solvers to produce concrete input values satisfying the identified constraints. This approach is argued to generate more powerful and effective MRs compared to traditional methods. However, it should be noted that using constraint solving requires significant technical expertise and resources. Additionally, constraint solving can be computationally expensive and may require longer processing times, particularly for complex systems.

In assessing the effectiveness of MRs, Chen *et al.* [75] conducted case studies applying MT to implementations of shortest path and critical path algorithms.

Their findings suggested that merely relying on theoretical properties is insufficient for distinguishing good MRs; instead, effective MRs are those that induce greater differences in the executions of the SUT. They proposed understanding the algorithm of the SUT before selecting MRs. Pioneering work by Asrafi *et al.* [38] offers a theoretical evaluation of MR effectiveness primarily through mutation analysis and code coverage. However, these metrics may not always provide actionable insights in specific industrial applications. Another study by Jia *et al.* [39] proposed a framework for assessing the effectiveness of MT. This involved using collections of real images from published libraries for generating test inputs, followed by mutation testing to evaluate fault detection rates. Their experiments showed that MT could detect faulty edge detection programs with up to 90% accuracy.

Jameel *et al.*[83] implemented Support Vector Machine (SVM) classifiers to automatically determine the correctness of output images, eliminating the need for manual validation in image processing applications. Their comparative study against traditional MT and statistical oracle methods demonstrated SVM's superiority in terms of classification error rates. In another study, Jafari *et al.* [40] selected existing MRs specific to image processing operations and proposed new MRs to fill identified gaps, particularly focusing on operations like dilation and erosion. They evaluated the fault detection rates of these MRs using mutation testing and compared them with existing ones to assess improvements or deficiencies. This comparison aids in understanding the robustness of MRs across different image processing tasks.

It is evident that mutation testing is a common method for assessing effectiveness; however, our study distinguishes itself by integrating MetaTrimmer, mutation testing, and three-level assessment strategy into a structured three-phase approach.

5. METATRIMMER: A TEST-DATA-DRIVING APPROACH FOR CLASSIFYING METAMORPHIC RELATIONS

Chapter 5, covers Contribution No. 1 and is based on the publications I, II, III and IV. The chapter is organised as follows: Section 5.1 provides the context of Contribution No. 1, detailing the background and challenges of MR classification, along with the motivation for proposing a TD-driven approach. Section 3.5 presents an overview of PMR approach, including its pipeline, dataset, predefined MRs, and evaluation results. In Section 5.2, we outline the key insights from the conceptual replication and extension of the PMR approach, establishing a foundation for comparison. Section 5.3 introduces MetaTrimmer, providing an in-depth explanation of its methodology and the results of the proof-of-concept evaluation. Finally, Section 5.3.5 discusses the answers to the research questions.

5.1. Introduction

One of the earliest and most influential works in generating MRs following a syntactically-based approach is PMR introduced by Kanewala *et al.* [29], published in STVR [29] and at ISSRE [30]. The idea behind PMR approach is to build a model to predict whether a method in a newly developed SUT can be tested using a specific MR. This approach relies on a predefined set of MRs and a classifier trained on CFG features extracted from a pool of sample methods. Initial evaluations of PMR used path- and node-based features extracted from CFG of Java methods and three predefined MRs to train SVM and decision tree models [30]. The approach was later extended to six predefined MRs using graph similarity measures, such as Random-Walk Kernel (RWK) and Gaussian Kernel (GK) [29]. Subsequent works have built on the PMR approach, including studies by Hardin *et al.* who aplied semi-supervised learning on CFG-based features [31], Rahman *et al.* who adopted PMR for matrix-based programs [22], and Zhang *et al.* [32] who introduced RBF-MLMR, a multi-label method for predicting MRs using radial basis function neural networks. Unlike PMR, which employs multiple binary classifiers, RBF-MLMR predicts all possible MRs for a given method. While RBF-MLMR uses a different algorithm than PMR, it adheres the same PMR methodology and feature extraction process.

Despite promising results from the PMR study and subsequent works, the PMR approach has significant limitations. First, it relies on binary classifiers that require labelled datasets for learning. However, labelled datasets are not always readily available, and creating them can be time-consuming. Second, the feature extraction process for model training is based on CFG, which may not account for refactoring. This limitation can impact the accuracy of the PMR since refactoring can alter code structure and, consequently, affect the applicability of MRs. Lastly,

the binary output of PMR does not consider TD and its impact on MRs applicability. This suggests that PMR does not account for cases where an MR may apply to specific TD characteristics while failing for others, leading to potential false positives or false negatives in MR selection.

To address these challenges, we propose MetaTrimmer, a TD-driven approach that shifts the focus from classification models to leveraging the relations between TD characteristics and MR violations. Building on insights from the PMR methodology, MetaTrimmer introduces a more flexible classification process that considers the behaviour of TD during the MT process. Instead of relying solely on code structure and CFG features, MetaTrimmer evaluates MRs based on their applicability across diverse TD inputs, recognising that MRs may not universally apply to all input spaces.

In this chapter, we address RG_1, which aims to propose a method for classifying MRs based on TD. The chapter answers the following research questions:

- *$RQ_{1.1}$: How can TD be used to classify MRs?*
- *$RQ_{1.2}$: How well does the TD-driven MR classification method perform?*

To address these questions and achieve the RG_1, we first revisit and replicate the original PMR approach, reconstructing the preprocessing and training pipeline to evaluate its performance across different programming languages, including Java, Python, and C++. Our replication confirmed that while PMR is effective for Java, its performance declines significantly when applied to functionally identical methods in other languages without retraining the classifiers. This highlighted the need for a more adaptable approach that can transcend language-specific constraints. Additionally, we extended the original PMR approach by considering features beyond CFGs, defining 21 source code-based features and building several classifiers. Our results show that while the original CFG-based features—particularly with an SVM using the RWK—yield better predictions for most MRs, our source code-based features achieved the best AUC-ROC (greater than 0.8) for one candidate MR.

Building on the findings of the replication study and the exploration of new features for training the classifiers, we propose a shift toward a TD-centric approach. MetaTrimmer introduces a process where TD is transformed based on the MRs, executed, and analysed to classify MRs based on their observed violation status. Through this approach, we discovered that some MRs are not universally applicable across all TD inputs, leading to the concept of mixed cases—scenarios where MR violations are not consistently observed. These mixed cases underscore the importance of considering the variability within TD when determining MR applicability. Through comparative analysis with the PMR approach, we demonstrate that MetaTrimmer offers a more refined and flexible method for MR classification.

5.2. Replication and Extension of the PMR Approach: Key Insights and Findings

In this section, we provide an overview of a conceptual replication study (Section 5.2.1) and the subsequent extension of the PMR approach (Section 5.2.2). We then highlight the key findings and the main outcomes of these efforts (Section 5.2.3).

5.2.1. Conceptual Replication of PMR Approach

We follow the guidelines suggested by Carver [84] and the ACM guidelines on reproducibility (different team, different experimental setup) [85]: *"The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same or similar result using artefacts, which they develop completely independently"*. We decided to replicate the Kanewala *et al.* [29] PMR study in three steps. In each step, we use the same set of predefined MRs used in the original study.

In the first step, we rebuilt the entire pipeline for extracting CFG information from source code to training and testing classification models in order to repeat the process described in the original study by Kanewala *et al.* The authors of the original study provided the extracted CFG information for replicating their work but not the Java source code of the analysed methods, which we retrieved from the corresponding project repositories on GitHub. We also had to develop the classification models that we then used for predicting MRs, because also they were not shared by the authors of the original paper. We conducted the first step to check whether we can re-create the classifiers with as good quality as in the original study and to prepare for the next steps by building our own classifiers based on features extracted from the CFG derived from the source code of methods.

The next two steps of our study aimed at exploring the transferability and generalisability of the PMR method. In the second step, we checked whether classifiers generated from Java code perform equally well when applied to Python and C++ code. Therefore, we created two datasets, one comprising 100 methods in Python and other one comprising 100 methods in C++. Both sets of methods implemented the exact same functionality as the 100 Java methods used in the first step. We did this to guarantee that the MRs taken from the original study would apply in the same way to the Python and C++ methods as they did in the original study using Java. To check the generalisability of the PMR method to other programming languages we then also developed individual classifiers for each programming language (Python, C++) starting out from code in the same programming language to which the classifier will be applied during evaluation.

The results of our study indicate that PMR classifiers generated based on artefacts implemented in one programming language do not perform well when applied to artefacts implemented in a different programming language, even though

the classifiers are based on CFGs to abstract from programming language and implementation details, the implemented functionality is exactly identical, and the set of MRs remains unchanged. On the other hand, it seems to be possible to generalise the PMR method in the sense that it can be applied with good performance on code implemented in a different programming language as long as the PMR classifiers are redeveloped based on code implemented in the same programming language.

The detailed replication study, including the methodology, research questions, and implementation details, can be found in Appendix A, as well as in [35] (publication I).

5.2.2. Extension of PMR Approach

The key part of PMR is *feature design*. In the original PMR work [30], Kanewala *et al.* used the CFG's path- and node-based features of 48 Java methods and three predefined MRs to train SVM and decision trees models. The authors show encouraging results when using node- and path-based features and SVM. Then, Kanewala *et al.* [29] extends their initial work by examining 100 Java methods and a set of six predefined MRs. As in the initial study, Kanewala *et al.* use SVMs, and features extracted from the methods' CFGs. However, instead of the node- and path-based features, the authors used measures of similarity between graphs. In particular, RWK and GK. Kanewala *et al.* concluded that SVM models trained with RWK-based features performed better than those trained with GK and with node- and path-based features (using a default linear kernel).

While the original study by Kanewala *et al.* [29], [30] showed encouraging results, the design of PMR features was limited to features extracted from the methods' CFGs only. First generating CFGs from the source code and then extracting features is relatively costly as compared to extracting features directly from the source code. To see whether and how the PMR approach could be improved by using features extracted directly from source code, as well as using those features with other classification approaches than SVM, we decided to conduct a study using the same set of methods and the same set of MRs as in Kanewala *et al.* [29]. Therefore, we extended the original PMR approach by looking at 21 features directly extracted from the source code. In addition, we experiment with five different binary classification models (including SVM).

Our results indicate that using the original CFG-based method-level features, in particular for a SVM with RWK, achieve better predictions in terms of AUC-ROC for most of the candidate MRs than our models. However, for one of the candidate MRs, using source code features achieved the best AUC-ROC result (greater than 0.8).

The detailed extension study, including the methodology, research questions, and implementation details, can be found in Appendix B, as well as in [7] (publication II).

5.2.3. Key Takeaways from Replication and Extension

The baseline approach for selecting MRs from a predefined set is PMR proposed by Kanewala *et al.* [29]. As previously discussed, the idea behind PMR is to develop a model capable of predicting whether a specific MR can be used to test a method in a newly developed SUT. To evaluate the generalisability of PMR across various programming languages, we conducted a replication study on PMR [35]. Our replication study involved reconstructing the preprocessing and training pipeline. The results of our replication study validated the reported findings and laid the groundwork for subsequent experiments.

In addition to this, we explored the potential reusability of the PMR model initially trained on Java methods. We assessed its suitability for functionally identical methods implemented in Python and C++. While the PMR model demonstrated strong performance with Java methods, its prediction accuracy notably declined when applied to Python and C++ methods. Nevertheless, we found that retraining the classifiers using CFGs specific to Python and C++ methods led to improved performance. Furthermore, we conducted an evaluation of the PMR approach, considering source code metrics as an alternative to CFG for building the models [7]. Our results also led to the conclusion that a generalisation of PMR beyond unit testing, such as its application to system-level testing, does not appear to be feasible.

5.3. MetaTrimmer

Despite promising results from the PMR study and subsequent works (*e.g.,* [30], [65], [29], [31], [22], [80], [32]), the PMR approach has significant limitations. Firstly, it relies on binary classifiers that require labelled datasets to provide examples for learning. Labelled datasets may not always be available, and obtaining them can be time-consuming. Secondly, the feature extraction process for model training is based on CFG or source code metrics, which may not account for refactoring. This limitation can affect the accuracy of the PMR approach, as refactoring can change the structure of the code and, consequently, the way MRs apply. Lastly, the binary output of PMR may not consider TD and its impact on MR applicability. This limitation implies that PMR does not consider the possibility that an MR may apply to some TD with specific characteristics and not others, leading to false positives or false negatives in MR selection.

Motivated by the limitations of PMR, we proposes MetaTrimmer, a TD-driven method for classifying MRs. Similar to PMR, we assume a predefined list of MRs is available. However, MetaTrimmer does not rely on labelled datasets and takes into account that an MR may only be applicable to TDwith specific characteristics.

This section, therefore, delves into the details of MetaTrimmer (Section 5.3.1), its evaluation and results (Section 5.3.3), threats to validity of the evaluation (Section 5.3.4) and the key findings (Section 5.3.5).

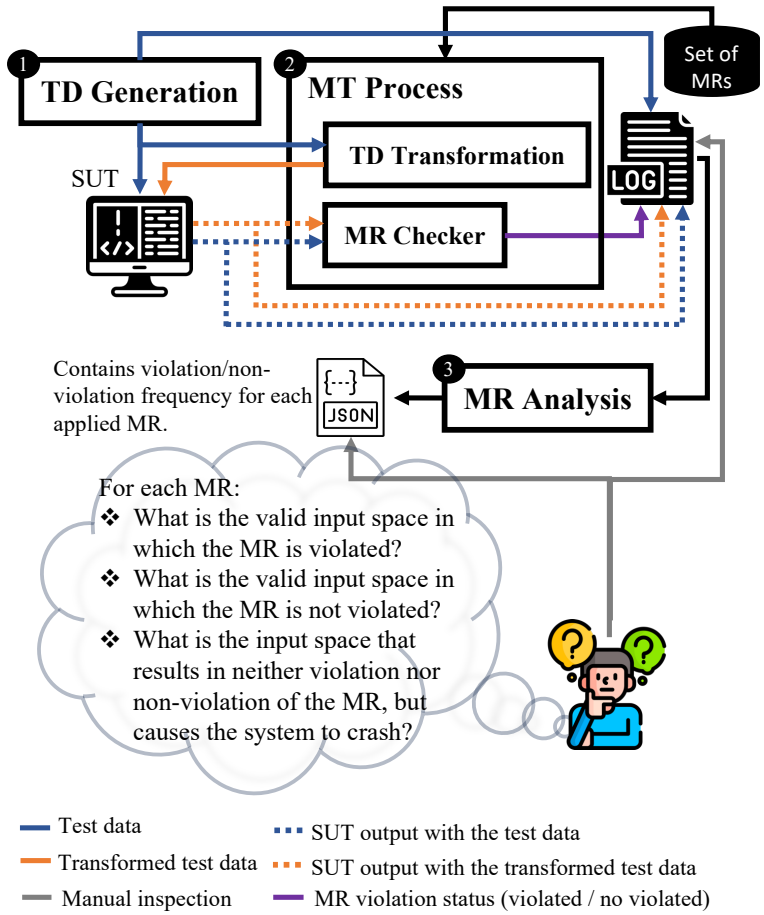

Figure 8. MetaTrimmer overview workflow

5.3.1. MetaTrimmer Process

Figure 8 provides an overview of MetaTrimmer. Overall, MetaTrimmer consists of three main steps: TD Generation, MT Process, and MR Analysis. In the TD Generation step, TD is generated. The MT Process step consists of three internal processes. Firstly, the TD produced in the TD Generation step is transformed based on the MRs specifications, resulting in TTD. Next, both the TD and TTD are executed against the SUT, and the outputs are compared against the MR specifications using the MR Checker, which provides a violation status for each MR. Throughout these steps, the TD, TTD, corresponding SUT outputs, and the violation status of each predefined MR are stored in a log file. Finally, in the MR Analysis step, the recorded data is analysed to determine the frequency of MR violations and non-violations for each applied MR. The output of this step is a `JSON` file that contains such information.

The selection and constraint process involves manual analysis and inspection. The frequency of MR violations or non-violations is examined to determine their applicability to the SUT. A consistent 100% violation rate indicates that the MR does not apply to the SUT, while a non-violation rate suggests a match with the SUT behaviour. In scenarios with mixed cases, *i.e.,* where violations and non-violations are not 100%, it becomes crucial to identify specific TD or ranges where the MR is applicable. This situation raises important questions for the tester: *'What is the valid input space in which the MR is violated or not violated?'* and *'What is the input space that results neither violates nor non-violation of MRs, but causes the system to crash?'*. By considering these questions and performing the necessary analysis, the tester can not only select MRs but also accurately specify the input-output relations across the valid input space.

5.3.2. SUT and Predefined Set of MRs

As we explain in Section 5.2, Kanewala *et al.* manually labelled 100 Java methods with a set of six predefined MRs in a binary manner. Then, in [35], we extended the original PMR approach to include two additional programming languages, Python and C++. We created two datasets consisting of source codes of methods written in Python and C++. The methods in each dataset are functionally identical to those used in the original Java dataset used by Kanewala *et al.* Because the functionality of the Python methods is equivalent to that of the Java methods, the same MRs that matched the Java methods would also match the corresponding Python and C++ methods.

Table 7. MRs used and the total number of methods to which a specific MR applies (column '✓')

MR	Change in the input	Output expected	✓	✗
MR_{PER}	Permute the components	Remain constant	23	2
MR_{ADD}	Add a positive constant	Increase or remain constant	21	4
MR_{MUL}	Multiply by a positive constant	Increase or remain constant	24	1
MR_{INV}	Take the inverse of each element	Decrease or remain constant	20	5
MR_{INC}	Add a new element	Increase or remain constant	14	11
MR_{EXC}	Remove an element	Decrease or remain constant	13	12
Total number of cases to which the set of MRs applies (✓) and does not apply (✗)			115	35

The MetaTrimmer evaluation used 25 Python methods from the dataset we created in [35]. We also kept the same MRs and their labels, ensuring they are the same as those created by Kanewala *et al.*[29]. Table 7 summarises the MRs used, the changes in the inputs and expected outputs, and the total number of methods to which a specific MR applies. Table 8 presents the list of methods with their respective labels as reported in [29] and [35]. The label '1' indicates that the MR applies (always), while the label '0' indicates that the MR does not apply (always) to the method.

Table 8. Set of methods with labels from [29] and [35]: '1' denotes the MR-method applies (always); '0' denotes that the MR-method does not applies (always)

Method name	$\mathbf{MR}_{PER}$	$\mathbf{MR}_{ADD}$	$\mathbf{MR}_{MUL}$	$\mathbf{MR}_{INV}$	$\mathbf{MR}_{INC}$	$\mathbf{MR}_{EXC}$
add_values	1	1	1	1	1	1
average	1	1	1	1	0	0
checkNonNegative	1	0*	1	1	1	0
checkPositive	1	0*	1	1	1	0
cnt_zeros	1	0*	0*	0*	1	1
count_non_zeros	1	1	1	1	1	1
durbinWatson	0	0	1	0	0	0
entropy	1	1	1	0	1	1
find_magnitude	1	1	1	1	1	1
find_max	1	1	1	1	1	1
find_max2	0	1	1	1	1	1
find_median	1	1	1	1	0	0
find_min	1	1	1	1	0	1
geometric_mean	1	1	1	1	0	0
harmonicMean	1	1	1	1	0	0
kurtosis	1	1	1	0	0	0
max	1	1	1	1	1	1
min	1	1	1	1	0	0
product	1	1	1	1	1	1
safeNorm	1	1	1	1	1	1
sampleVariance	1	1	1	1	0	0
skew	1	1	1	0	0	0
sum	1	1	1	1	1	1
sumOfLogarithms	1	1	1	1	1	1
variance	1	1	1	1	0	0

5.3.3. Evaluation and Results

For the evaluation of MetaTrimmer, we consider two scenarios. In the first scenario, we assume prior knowledge of the initial restriction on the TD, specifically that only positive numbers are allowed. In the second scenario, we explore the case where this prior knowledge is absent.

Figure 9 illustrates the pipeline that was developed for implementing MetaTrimmer and conducting the evaluations. The implementation consists of five Python scripts. The first script, named `InputGenerator.py`(1), is a fuzzer that uses a random number generator to create the TD, which is based on a list of parameters. The output of this script is a `JSON`(a) file that contains the generated TD. The parameters are:

- low (`l`): Fuzzer minimum value threshold. The minimum number that the fuzzer will consider. For instance, if “low" is set to -1, the fuzzer will not generate elements less than -1.
- high (`h`): Fuzzer maximum value threshold. The maximum number the fuzzer will consider. For instance, if “high" is set to 5, the fuzzer will not generate elements greater than 5.
- input_type (`it`): The type of elements in the list. Currently, the only available options are ‘int’ or ‘float’.
- t_{end} (`t`): The end time of the program execution.
- output (`o`): The name of the output file.

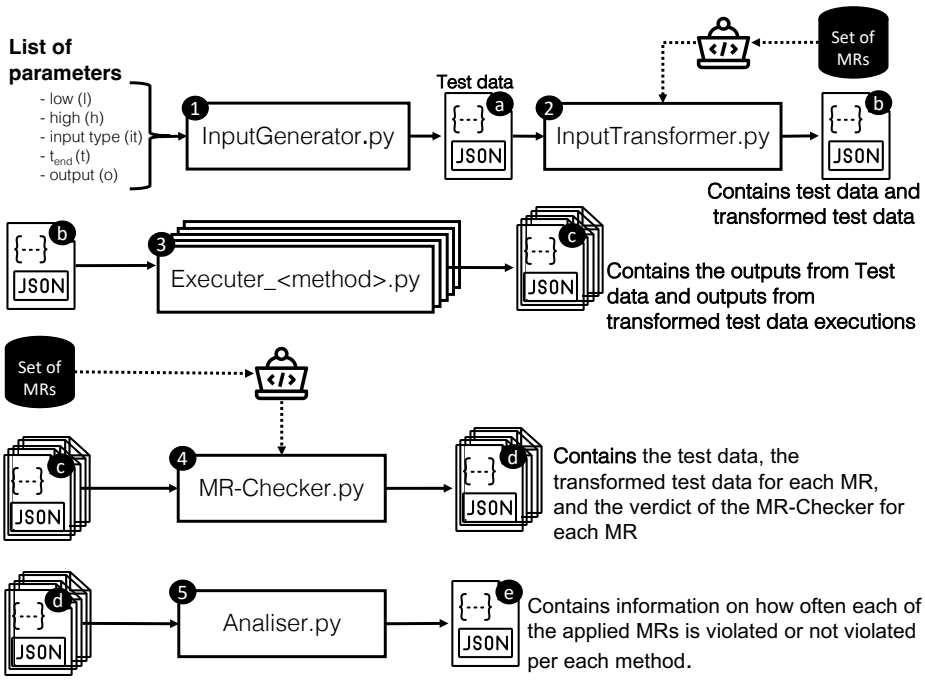

Figure 9. Possible implementation pipeline of MetaTrimmer.

The second script, `InputTransformer.py`❷, transforms the TD generated by `InputGenerator.py` based on the rules defined by each MR. The script named `InputTransformer.py` takes the `JSON`ⓐ file containing the generated TD as input and produces a new `JSON`ⓑ file with the transformed data as output. The number of fields in the `JSON`ⓑ file will depend on the number of MRs used. For instance, we used six MRs, so we expected to have seven fields: the original TD and the transformed data for each of the six MRs. It is important to note that the MRs are described in natural language and must be translated into code to be applied to the TD. The MRs are hardcoded in `InputTransformer.py` script so that they are applied automatically to the generated TD to generate the TTD.

As Figure 9 shows, for each method tested, there exists an `script`③ script. The main purpose of the executer scripts is to run the original TD, and the transformed TD generated for each MR, *i.e.,* `JSON`ⓑ, and store their outputs, represented by output `JSON`ⓓ in the Figure 9. The number of executers required depends on the number of methods being tested. In our experiment, we tested 25 different methods, creating 25 different executers and getting 25 `JSON`ⓒ files.

Then, the `MR-Checker.py`④ script consumes the outputs generated by the executer scripts and checks whether the outputs of the original TD and the transformed TD match the corresponding MR. Similar to `InputTransformer.py`, the expected output relations for each MR are hardcoded into the MR-Checker script, which means that the script applies these relations automatically to the generated outputs from the executer scripts. After the TD is generated, transformed, and checked, a `JSON`ⓓ file is produced per each method containing information about the execution ID, the original TD, the transformed TD for each MR, and the verdict of the MR-Checker for each MR. The `JSON`ⓓ files produced by the MR-Checker are used as input for the final script, called `Analiser.py`⑤. The `Analiser.py` script's purpose is to calculate for each method how often each of the applied MRs is violated or not violated, and store this information, `JSON`ⓔ in Figure 9.

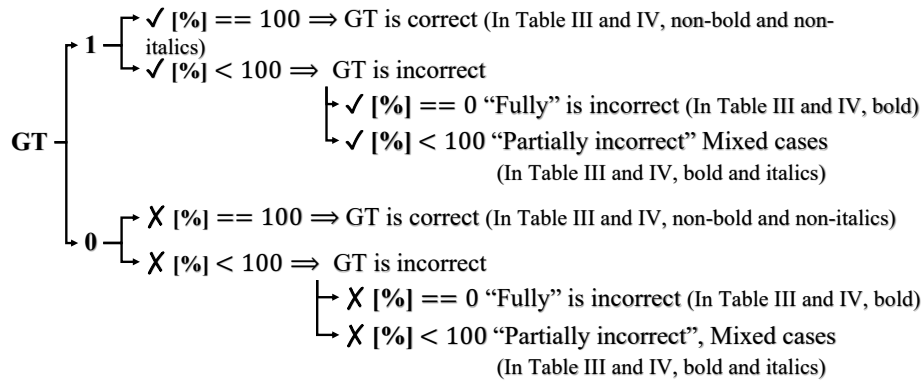

Figure 10. GT values for MetaTrimmer evaluation: 1 means the MR always applies, while 0 means it doesn't always apply. We use symbols ✓ and ✗ to indicate the percentage of runs where the MR applies or is violated. ✓ at 100% means the GT is correct. Otherwise, it's incorrect. If ✓ is 0, GT is fully incorrect; if it's less than 100% but not 0, it's partially incorrect (mixed case). Similarly, ✗ at 100% means the GT is correct. Otherwise, it could be partially incorrect (mixed case), but if ✗ is 0, the GT is incorrect.

An important aspect of the MetaTrimmer evaluation is understanding how well MetaTrimmer performs compared to the PMR approach in terms of MR classification. Specifically, we are interested in determining whether an MR applies to a specific method. We hypothesise that if the MR is violated in all TD, it does not apply to the tested method. Conversely, if the MR is not violated for all TD, it

applies to the tested method. MetaTrimmer differs from the PMR method in that it chooses MRs based on TD, while PMR employs binary classifiers to predict if a given method consistently applies to a specific MR. Thus, comparing the performance of the proposed method with the PMR approach in terms of MR selection is not appropriate. However, we can compare the outcomes of MetaTrimmer with the PMR approach in terms of the number of MRs that apply to a given method based on its Ground Truth (GT). We assumed that the GT labels were intended for only positive numbers. As such, we generated TD with the constraint of using positive numbers only, and we avoided any possible exceptions that may arise due to invalid inputs. To generate the TD, we used `InputTransformer.py` script with the parameters specified below: `l`= 1, `h`= 50, `it`= int, `t`= 0.5 s.

Table 9. Set of methods with the GT from [29] and [35]: '1' means that the MR always applies, and '0' means that the MR does not always apply. Symbol ✓ denotes the percentage of runs when the MR applies, and symbol ✗ denotes the percentage of runs when the MR does not apply (= is violated). Test data restriction: only positive integer numbers

Method name	MR_{PER}			MR_{ADD}			MR_{MUL}			MR_{INV}			MR_{INC}			MR_{EXC}		
	GT	✓ [%]	✗ [%]	**GT**	✓ [%]	✗ [%]	**GT**	✓ [%]	✗ [%]	**GT**	✓ [%]	✗ [%]	**GT**	✓ [%]	✗ [%]	**GT**	✓ [%]	✗ [%]
add_values	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
average	1	100	0	1	100	0	1	100	0	1	100	0	0	0	100	*0*	*50*	*50*
checkNonNegative	1	100	0	**0**	**100**	**0**	1	100	0	1	100	0	1	100	0	**0**	**100**	**0**
checkPositive	1	100	0	**0**	**100**	**0**	1	100	0	1	100	0	1	100	0	**0**	**100**	**0**
cnt_zeros	1	100	0	**0**	**100**	**0**	**0**	**100**	**0**	**0**	**100**	**0**	1	100	0	1	100	0
count_non_zeros	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
durbinWatson	*0*	*4*	*96*	0	0	100	1	100	0	*0*	*14*	*86*	*0*	*88*	*12*	*0*	*48*	*52*
entropy	1	100	0	1	100	0	1	100	0	*0*	*89*	*11*	1	100	0	1	100	0
find_magnitude	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
find_max	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
find_max2	*0*	*14*	*86*	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
find_median	1	100	0	1	100	0	1	100	0	1	100	0	*0*	*9*	*91*	*0*	*54*	*46*
find_min	1	100	0	1	100	0	1	100	0	1	100	0	*0*	*49*	*51*	**1**	*86*	*14*
geometric_mean	1	100	0	1	100	0	1	100	0	1	100	0	0	0	100	*0*	*59*	*41*
harmonicMean	1	100	0	1	100	0	1	100	0	1	100	0	*0*	*3*	*97*	*0*	*69*	*31*
kurtosis	1	100	0	1	100	0	1	100	0	*0*	*17*	*83*	*0*	*48*	*52*	*0*	*65*	*35*
max	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
min	1	100	0	1	100	0	1	100	0	1	100	0	*0*	*49*	*51*	*0*	*86*	*14*
product	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
safeNorm	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
sampleVariance	1	100	0	1	100	0	1	100	0	1	100	0	0	0	100	0	0	100
skew	1	100	0	1	100	0	1	100	0	*0*	*11*	*89*	*0*	*81*	*19*	*0*	*50*	*50*
sum	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
sumOfLogarithms	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0	1	100	0
variance	1	100	0	1	100	0	1	100	0	1	100	0	*0*	*91*	*9*	*0*	*46*	*54*

GT: Ground Truth

Figure 10 shows the possible values of GT and its corresponding meaning for MetaTrimmer. When GT is set to '1', the MR always applies. On the other hand, a value of '0' means that the MR does not always apply. The symbol ✓ denotes the percentage of runs when the MR applies, and it has two possible outcomes: If ✓ is 100%, we assume that the GT is correct. These cases are marked in Table 9 and Table 10 using non-bold and non-italics formatting. However, if it is less than 100%, we assume the GT is incorrect. In such cases, we distinguish between two types of incorrectness: if ✓ is equal to 0, it means that the GT is fully incorrect. These cases are marked in Table 9 and Table 10. If ✓ is less than 100% but not

equal to 0, we refer to it as a mixed case, where the GT is partially incorrect. These cases are marked in Table 9 and Table 10 using bold and italics formatting. Similarly, the symbol ✗ denotes the percentage of runs where the MR does not apply or is violated. If ✗ is 100%, we assume that the GT is correct. These cases are marked in Table 9 and Table 10 using non-bold and non-italics formatting. However, if it is less than 100%, we may consider it a mixed case, where the GT could be partially incorrect. These cases are marked in Table 9 and Table 10 using italics formatting. If ✗ is equal to 0, it means that the GT is incorrect. These cases are marked in Table 9 and Table 10 using bold formatting.

Table 9 shows the methods used, their GT, and the frequency with which each applied MR is violated or not violated per method. While analysing Table 9, at least four distinct patterns can be drawn: *i)* Methods labelled as 1 in the GT showed a 100% compliance rate, indicating that the MRs may apply to those methods. *ii)* In only 5 out of 35 cases, an MR was violated 100% of the time. This is for, `durbinWatson` violated MR_{ADD} 100% of the time, while `average`, `geometric_mean`, and `sampleVariance` violated MR_{INC} and `sampleVariance` violated MR_{EXC}, all 100% of the time. *iii)* Some methods had a small proportion of non-violations (less than 17%) and a high proportion of violations. This was observed in 7 out of 35 cases. For instance, `durbinWatson` violated MR_{PER} only 4% of the time, while `harmonic_mean` violated it only 3% of the time.

These findings suggest that there is not always a scenario where an MR cannot apply to a specific method. However, it is important to note that such findings do not render the MRs useless. Instead, investigating and understanding the small proportion of non-violations can be valuable in improving the MRs. By making them more precise and customised to the specific TD. *iv)* the analysis revealed cases where violations and non-violations were relatively proportional. This was observed in 11 out of 34 cases, where the violations and non-violations were proportional, indicating that the MRs but with exceptions. For instance, `kurtosis` violated MR_{INC} 52% of the time and had no violations 48% of the time, while for MR_{EXC}, it had 35% violations and 65% non-violations. This implies that exceptions must be considered and accounted for when applying MRs. In such cases, defining constraints based on the specific TD is even more important.

In the previous scenario, we assumed a scenario where we had prior knowledge of the initial restriction for the TD, *i.e.,* only positive numbers. Now, we investigate a scenario where we lack this prior knowledge. Thus, we aim to determine if mixed cases can provide valuable information for constraining MRs. To generate the TD, we used `InputTransformer.py` script with the parameters specified below: `l`= -15, `h`= 15, `it`= int, `t`= 0.5 s.

Upon initial examination of the results presented in Table 10, one can observe three types of outcomes: violation, non-violation, and invalid data. The percentage of invalid data is denoted in parentheses in the column that indicates that the MR does not apply (*i.e.,* column ✗ [%]). Upon analysing the results presented in Table 10, one can identify at least five patterns. *i)* It is evident that the applicabil-

Table 10. Set of methods with the GT from [29] and [35]: '1' means that the MR always applies, and '0' means that the MR does not always apply. Symbol ✓ denotes the percentage of runs when the MR applies, and symbol ✗ denotes the percentage of runs when the MR does not apply. Test data restriction: only integers. Numbers in parentheses refer to the percentage of invalid input data (crashes).

Method name	MR_{PER}			MR_{ADD}			MR_{MUL}			MR_{INV}			MR_{INC}			MR_{EXC}		
	GT	✓ [%]	✗ [%]	GT	✓ [%]	✗ [%]	GT	✓ [%]	✗ [%]	GT	✓ [%]	✗ [%]	GT	✓ [%]	✗ [%]	GT	✓ [%]	✗ [%]
add_values	1	100	0	1	100	0	***1***	***46***	***54***	***1***	***37***	***44* (19)**	1	100	0	***1***	***50***	***50***
average	1	100	0	1	100	0	***1***	***46***	***54***	***1***	***37***	***44* (19)**	*0*	*80*	*20*	*0*	*50*	*50*
checkNonNegative	1	100	0	**0**	**100**	**0**	1	100	0	***1***	***81***	***0* (19)**	1	100	0	**0**	**98**	**2**
checkPositive	1	100	0	**0**	**100**	**0**	1	100	0	***1***	***81***	***0* (19)**	1	100	0	**0**	**98**	**2**
cnt_zeros	1	100	0	**0**	**84**	**16**	**0**	**100**	**0**	**0**	**81**	**0 (19)**	1	100	0	1	100	0
count_non_zeros	1	100	0	***1***	***84***	***16***	1	100	0	***1***	***81***	***0* (19)**	1	100	0	1	100	0
durbinWatson	*0*	*4*	*96*	**0**	*41*	*59*	1	100	0	*0*	*39*	*42* (19)	*0*	*80*	*20*	*0*	*55*	*45*
entropy	1	100	0	1	100	0	1	100	0	*0*	*58*	*24* (18)	1	100	0	1	100	0
find_magnitude	1	100	0	***1***	***60***	***40***	1	100	0	***1***	***81***	***0* (19)**	1	100	0	1	100	0
find_max	1	100	0	1	100	0	***1***	***98***	***2***	***1***	***79***	***2* (19)**	1	100	0	1	100	0
find_max2	*0*	*25*	*75*	1	100	0	*1*	*92*	*8*	***1***	***73***	***8* (19)**	1	100	0	1	100	0
find_median	1	100	0	1	100	0	***1***	***49***	***51***	***1***	***38***	***43* (19)**	*0*	*75*	*25*	*0*	*55*	*45*
find_min	1	100	0	1	100	0	***1***	***2***	***98***	*0*	*2*	*79* (19)	***0***	***99***	***1***	***1***	***87***	***13***
geometric_mean	***1***	***59***	***0* (41)**	***1***	***24***	***19* (57)**	***1***	***59***	***0* (41)**	***1***	***40***	***0* (60)**	*0*	*19*	*40* (41)	*0*	*28*	*9* (63)
harmonicMean	***1***	***81***	***0* (19)**	***1***	***43***	***23* (34)**	***1***	***40***	***41* (19)**	***1***	***40***	***41* (19)**	*0*	*17*	*64 (19)*	*0*	*41*	*40 (19)*
kurtosis	1	100	0	1	100	0	1	100	0	*0*	*19*	*62*	*0*	*80*	*20*	*0*	*65*	*35*
max	1	100	0	1	100	0	***1***	***98***	***2***	***1***	***79***	***2* (19)**	1	100	0	1	100	0
min	1	100	0	1	100	0	***1***	***2***	***98***	***1***	***2***	***79* (19)**	*0*	*99*	*1*	*0*	*87*	*13*
product	1	100	0	***1***	***52***	***48***	***1***	***59***	***41***	***1***	***41***	***40* (19)**	***1***	***59***	***41***	***1***	***59***	***41***
safeNorm	1	100	0	***1***	***60***	***40***	1	100	0	***1***	***81***	***0* (19)**	1	100	0	1	100	0
sampleVariance	1	100	0	1	100	0	1	100	0	***1***	***81***	***0* (19)**	0	0	100	0	0	100
skew	1	100	0	1	100	0	1	100	0	*0*	*40*	*41 (19)*	*0*	*18*	*82*	*0*	*50*	*50*
sum	1	100	0	1	100	0	***1***	***46***	***54***	***1***	***37***	***44* (19)**	1	100	0	***1***	***50***	***50***
sumOfLogarithms	***1***	***81***	***0* (19)**	***1***	***34***	***32* (34)**	***1***	***81***	***0* (19)**	***1***	***81***	***0* (19)**	***1***	***81***	***0* (19)**	***1***	***81***	***0* (19)**
variance	1	100	0	1	100	0	1	100	0	***1***	***81***	***0* (19)**	*0*	*9*	*91*	*0*	*46*	*54*

GT: Ground Truth (NB: We use the same GT as in TABLE III to make the direct comparison of entries easier)

ity of MRs can be easily determined by checking if there are 100% no violations. When the GT is labelled as '1', we observe that some previously identified MRs still hold. For example, 20 out of the 23 cases in MR_{PER} remain unchanged.

ii) Unlike in Table 9, some GTs marked as '1' in Table 10 have mixed cases. For instance, the `add_values` method under MR_{MUL} has 46 non-violations and 54 violations. *iii)* When there is no initial constraint, a third type of outcome could happen, invalid data. For instance, the `geometric_mean` method has only 59% non-violations and 0% violations, indicating that 41% of the TD was invalid. *iv)* Some methods had a small proportion of non-violations and a high proportion of violations and vice versa. *v)*, similar to Table 9, there are cases where the violations and non-violations were proportional. To extract constraints, we manually inspected the mixed cases for each method. This detailed examination allowed us to identify a clear pattern: most violations occurred when negative numbers were involved in the TD. We also noticed that empty arrays in the TD could result in either violations or invalid data.

5.3.4. Threats to Validity

In the context of MetaTrimmer, two types of threats to validity are most relevant: threats to internal and external validity.

To achieve internal validity, we used the same set of methods and MRs as in Kanewala *et al.* There is a potential risk of bias in selecting the MRs or the TDused, which could potentially impact the results obtained. For instance, we found cases where certain methods and MRs were incorrectly labelled during our analysis.

To enhance external validity and ensure the generalisability of the results, it would have been preferable to include a wider range of methods in the evaluation process. This would have helped to mitigate any potential bias that could have arisen from the selection of the methods used. Therefore, the current study may not be able to determine the full extent of the effectiveness of MetaTrimmer.

5.3.5. Key findings

In the first scenario, where prior knowledge of the initial restriction on the TD was assumed, our findings show that MetaTrimmer outperforms the PMR approach in classifying MRs, achieving a 100% compliance rate for all methods labeled as '1' in the GT. Additionally, we identified three methods with incorrect labels. MetaTrimmer also identified cases where MRs do not apply, cases with a small proportion of non-violations and a high proportion of violations, as well as cases where violations and non-violations are relatively balanced for methods labeled as '0' in the GT. These findings suggest that there is not always a scenario in which an MR cannot apply to a specific method. By investigating and understanding these non-violations, we can refine the MRs, making them more precise and tailored to the specific TD, *i.e.,* by adding constraints. Even with initial constraints for the TD (only positive integers), MetaTrimmer provides valuable insights into constraints for MRs that may only apply to specific TD. These constraints can help uncover situations that improve the overall test coverage of the test suite.

In the second scenario, in which prior knowledge is not possible, our findings suggest that TD can be leveraged to derive constraints for MRs by analysing their violations and non-violations across different inputs. When an MR consistently holds for specific TD, it can be considered a constraint for that MR. Conversely, if an MR is consistently violated for particular TD, it indicates that the MR may not be suitable for the SUT. Mixed cases may also arise, highlighting specific TD or TD ranges where the MR is applicable, thus providing constraints to narrow the MR's scope accordingly. By examining the reasons behind these mixed cases, we can identify specific constraints for the MR.

5.4. Discussion

In this chapter, we addressed RG_1, which aims to provide a method for classifying MRs based on TD. To achieve this, we began by exploring the PMR approach through a replication study and its subsequent extension. These efforts provided valuable insights and established a baseline for comparison. Based on our findings from these studies, we proposed a shift toward a TD-Driving approach for MR classification, leading to the development of MetaTrimmer.

MetaTrimmer builds on the premise that TD characteristics can be leveraged to dynamically classify MRs, moving beyond static code-based approaches such as PMR. Below, we discuss the answers to the research questions that guided this work:

5.4.1. $RQ_{1.1}$: How can TD be used to classify MRs?

To address this question, we developed MetaTrimmer, a method that utilises TD to evaluate the applicability of MRs. In MetaTrimmer, TD is transformed according to predefined MR specifications, and both the original and TTD are executed against the SUT. By analysing the outcomes—specifically, whether the MR is violated or not—MetaTrimmer classifies MRs based on their behaviour across diverse inputs.

This TD-driven process allows MetaTrimmer to go beyond binary classifications by identifying mixed cases, where an MR may be applicable to some TD inputs but not others. These mixed cases enable us to derive constraints for MRs, helping to define the exact conditions under which an MR holds. As a result, TD is not only used to classify MRs but also to refine them by specifying the precise input space where they apply, enhancing both the accuracy of MR selection and the effectiveness of the overall MT process.

5.4.2. $RQ_{1.2}$: How well does the TD-driven MR classification method perform?

In comparison to PMR, MetaTrimmer demonstrated promising results in handling mixed cases and dynamically adjusting MR applicability based on TD. In scenarios where prior TD constraints were known, MetaTrimmer achieved a 100% compliance rate for methods where MRs were expected to apply. Additionally, it identified several cases where methods were incorrectly labelled in the GT, providing evidence of its ability to detect nuanced MR behaviour.

When no prior TD constraints were provided, MetaTrimmer still derived meaningful insights by identifying the TD ranges where MRs consistently applied or failed. This flexibility in dealing with diverse inputs positions MetaTrimmer as a more adaptable and comprehensive approach compared to static methods like PMR.

5.5. Replication Packages and Artifacts

- Replication study
  https://github.com/aduquet/RENE-PredictingMetamorphicRelations
- PMR extension
  https://github.com/aduquet/VST22_PMR-SourceCodeMetrics
- SCminer for extracting source code metrics
  https://github.com/aduquet/SCminer

6. AN ASSOCIATION RULE MINING-BASED APPROACH FOR REFINING METAMORPHIC RELATIONS BASED ON TEST DATA

Chapter 6 covers Contribution No. 2 and is based on publications V and VII. The chapter is organised as follows: Section 6.1 introduces the context for Contribution No. 2, explaining the challenges associated with the manual inspection process when extracting constraints for MRs. It details the rationale behind refining MRs using constraints derived from TD analysis and provides the necessary background for understanding the proposed ARM-driven approach. Section 6.2 outlines the methodology by introducing the ARM-based pattern extraction module and how it enhances the MR Analysis step of MetaTrimmer. It also explains the experimental setup and evaluation approach, including the MRs and methods used as the SUT. Also, presents the evaluation results, demonstrating the performance of the ARM-based module in identifying patterns and refining MRs.

Section 6.3 provides the answers to the research questions by showing how ARM can be used to extract patterns and refine MRs. It highlights both the successes and challenges encountered during the analysis process. Section 6.4 provides links to the replication packages related to all the data and scrips generated during the experiments supporting Contribution No. 2.

6.1. Introduction

In the previous chapter, we introduced MetaTrimmer, a TD-driven approach for classifying MRs based on the behaviour of TD. MetaTrimmer acknowledges that MRs may not apply universally to the entire input data space but instead to specific subsets, referred to as constraints. In MetaTrimmer, an MR violated 100% of the time is considered inapplicable to the SUT, while an MR not violated 100% of the time is applicable. Cases where violations and non-violations occur inconsistently—referred to as mixed cases—offer valuable insights into the relationship between the SUT and specific MRs.

This challenges the logic of binary classifiers, which assume that a selected MR must always apply to the entire valid input data space. However, as we conclude from previous chapter, it is possible for an MR to apply to specific TD while not applying to others, leading to “false positives," where an MR violation incorrectly flags a fault in the SUT. Additionally, the expectation that an MR must always apply to the entire input space can limit the MR’s effectiveness. In real-world scenarios, it is common for different subsets of the input space to exhibit distinct behaviours and characteristics.

To illustrate this, let’s consider a program that calculates the average of elements in a list as the SUT, hereinafter referred to as `AVG`. The MRs that seems to apply are defined as follows:

- **MR$_{MUL}$: *IF*** *the inputs are multiplied by a positive constant K, where K* $>$ 1*,* ***THEN*** *the output must increase.*
- **MR$_{PER}$: *IF*** *the inputs are permuted,* ***THEN*** *the output must remain equal.*

Table 11 shows the initial TD in the column TD, the TD transformations specified by the MRs in the column TTD, and the respective outputs for the TD and TTD in the columns Output and Output-T. This table demonstrates the execution of the SUT with both the TD and TTD, and checks the expected behaviour in the outputs as indicated by the MRs. When computing the frequency of violations/non-violations, one can observe that MR$_{PER}$ presents a clear-cut scenario, with 100% non-violation across all input space. This indicates that MR$_{PER}$ can serve effectively as a regression test suite, covering the entire input space with no violations. However, for MR$_{MUL}$, the situation is not straightforward, with exactly 50% violations and 50% non-violations. Through manual inspection, one can conclude that the violations are not triggered by a fault in the SUT, but by the MR$_{MUL}$ itself. In traditional MT approaches, where only MRs that always apply (*i.e.,* 100% non-violations) are considered, MR$_{MUL}$ would not be included in the regression test suite. However, this can result in a less comprehensive regression test suite, potentially allowing defects to slip through undetected. By analysing these mixed cases, MetaTrimmer employs a manual process to inspect and understand the reasons behind the violation status (VS) of MRs. This analysis helps to identify patterns that can be used to set constraints, thereby refining the applicability of MRs. For example, by manually reviewing the data in Table 11, we can observe that non-violations occur when all the elements in the TD are positive, while violations occur when the elements are negative. These insights allow us to define more precise conditions under which the MR holds.

Table 11. Illustrative example (vs = 1: violation, vs = 0: non-violation)

TD	**Output**	**MR$_{MUL}$**			**MR$_{PER}$**		
		TDT	**Output-T**	**VS**	**TDT**	**Output-T**	**VS**
$[-11,4,-12,14]$	*-1.25*	$[-33,12,-36,42]$	*-3.75*	*1*	$[-4,11,14,-12]$	-1.25	0
$[3,2,18,0,16]$	7	$[9,6,54,0,48]$	23.4	0	$[0,3,16,2,18]$	7	0
$[-5,-1,14,1]$	2.25	$[-15,-3,42,3]$	9	0	$[15,-5,-1,1]$	2.25	0
$[-1,-2,-5]$	*-2.66*	$[-3,-6,-15]$	*-8*	*1*	$[-2,-5,-1]$	*-2.66*	0

However, while manual inspection offers valuable insights in MR Analysis, it may not always yield a completely accurate or comprehensive understanding of the underlying reasons behind the status of the MRs. This can result in extracting incomplete or incorrect constraints, especially when dealing with large amounts of TD or multiple MRs. For instance, in the `AVG` example, upon deeper inspection, one can notice that when the list contains both positive and negative numbers, if the sum of the positive numbers is greater than the sum of the negative numbers, it leads to a non-violation. Conversely, when the sum of the negative numbers is

greater, it leads to a violation of the MR.

In this chapter, we address RG_2, which aims to propose a method for refining MRs by setting constrains based on TD. The chapter answer the following research questions:

- ***RQ***2.1*: How can patterns be extracted and utilised as constraints to refine MRs?*
- ***RQ***2.2*: How well does the ARM-driven approach refine and constrain MRs?*

To address these questions, we extend the MetaTrimmer process by introducing an automated approach using ARM to assist with the pattern extraction in the MR Analysis step. ARM helps identify patterns in the TD and MR violation status, reducing reliance on manual inspection and enabling the definition of more constraints. By leveraging ARM, we aim to formalise the identification of specific TD characteristics that influence MR violations and refine MRs accordingly.

6.2. MetraTrimmer+

This section introduces the association-rule-based approach to support pattern extraction during the MR Analysis step of MetaTrimmer (Section 6.2.1), along with details of the experimental setup for evaluation. This encompasses information regarding the MRs, as well as the methods utilised as the SUT (Section 6.2.2). The evaluation and results (Section 6.2.3), threats to validity of the evaluation (Section 6.2.4).

6.2.1. Methodology

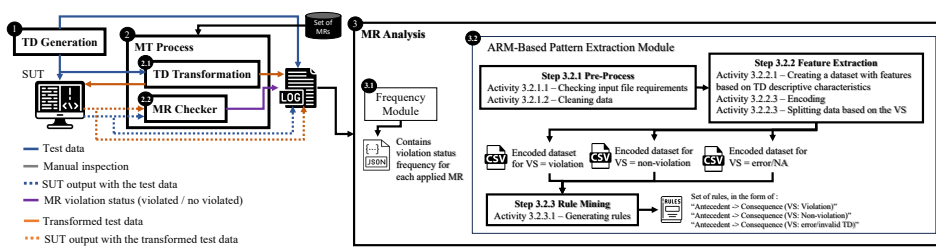

Figure 11. High-level MetaTrimmer overview workflow including the enhancement of the MR Analysis

The constraint definition process involves manual analysis and inspection in the MR Analysis step of MetaTrimmer. In the MR Analysis, the frequency of MR violations or non-violations is examined to determine their applicability to the SUT. A consistent 100% violation rate indicates that the MR does not apply to the SUT, while a non-violation rate of 100% suggests a match with the SUT behaviour. In scenarios with mixed cases, *i.e.,* where violations and non-violations

are not 100%, it becomes crucial to identify specific TD or ranges where the MR is applicable. This situation raises important questions for the tester: *'What is the valid input space in which the MR is violated or not violated?'* and *'What is the input space that results in neither violations nor non-violations of MRs, but causes the system to crash?, but causes the system to crash?'*. By considering these questions and performing the necessary analysis, the tester can select MRs and accurately specify the input-output relations across the valid input space. Figure 8 provides a high-level MetaTrimmer overview including the enhancement of the MR Analysis.

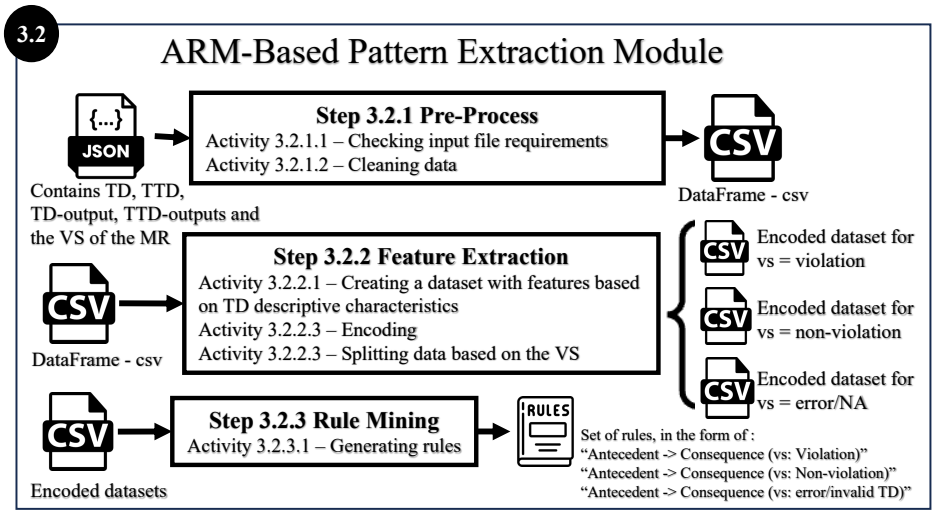

Figure 12. ARM-Based Pattern Extraction Module. The process is divided into three steps: Step 3.2.1 Pre-Process, which includes checking input file requirements and cleaning data; Step 3.2.2 Feature Extraction, which involves creating a dataset with features based on descriptive characteristics of the TD, encoding, and splitting data based on the violation status (VS); and Step 3.2.3 Rule Mining, which focuses on generating rules.

Figure 12 presents an overview of the ARM-Based Pattern Extraction Module. The process is divided into three steps: *Step 3.2.1 - Pre-Process*, which is responsible for checking input file requirements and cleaning data; *Step 3.2.2 - Feature Extraction*, which involves creating a dataset with features based on descriptive characteristics of the TD, encoding, and splitting data based on the VS, *i.e.,* violation, non-violation and in some cases, error/NA; and *Step 3.2.3 - Rule Mining*, which focuses on the rule generation. Below we describe them in detail:

Step 3.2.1 - Pre-Process: This step is responsible for checking the log's completeness and consistency, potential errors, invalid formatting, and prepare the data for the further steps. Completeness and consistency validate the integrity of the log data and ensure that it adheres to the expected structure. This pre-processing step is made up of two activities: *Activity 3.2.1.1 (Checking Input File Requirements)* is responsible for checking and formatting the data in the log. The input for

this activity is the log produced by the MT process, which contains information on TD, TTD, their respective outputs, and the VS. During this activity, checking the consistency in the nomenclature used for identifying column names, hereinafter referred to as attributes, is the priority. Additionally, this activity transforms the `JSON` object into a DataFrame.

Activity 3.2.1.2 (Cleaning Data) is tasked with cleaning up the data by removing inconsistencies. Given that the goal of the ARM-based approach is to extract patterns relating TD with VS, any information regarding TTD, TD, and TTD outputs that is not required is removed.

Step 3.2.2 - Feature Extraction: This step is responsible for creating a dataset with features based on TD descriptive characteristics, encoding the generated features, and splitting the dataset based on the VS when necessary. The feature extraction process is carried out through three activities, described below:

Activity 3.2.2.1 (Creating a Dataset with Features Based on TD Descriptive Characteristics) is responsible for extracting descriptive characteristics of the TD. This is a critical activity, as its primary goal is to characterise and simplify the TD by focusing on its qualitative and descriptive attributes. By generalising specific aspects of the TD, this activity aims to create a more organised and integrated dataset. This generalisation enhances the ability to identify similar patterns, thereby increasing the effectiveness of pattern extraction. To accomplish this task, it's essential to consider several factors, including the number of SUT inputs, the types of inputs, and their relations. For example, let's consider a scenario where the SUT has two integer inputs, TD_a and TD_b. Describing and generalising these inputs, regardless of their specific values, can be achieved using partial order theory.

Partial order defines a notion of comparison between at least two elements. In this case, one can establish relationships such as $TD_a < TD_b$, $TD_a = TD_b$, or $TD_a > TD_b$. These comparisons allow us to categorise and generalise the inputs based on their relationships, rather than solely on their individual values. By doing so, we can create a more abstract representation of the inputs, which facilitates pattern recognition and rule generation during the analysis process.

Activity 3.2.2.2 (Encoding) is in charge of preparing the data according to the requirements of the rule mining algorithm. For example, Apriori [86], which is the algorithm used in this contribution, works only with categorical features.

Activity 3.2.2.3 (Splitting Data Based on the VS) is responsible for splitting the dataset based on the VS. It is important to keep in mind that the logs analysed by the ARM-based pattern extraction module are mixed cases, meaning those where there is no clear distinction regarding the applicability of the MR, as the samples violating and non-violating the MR are balanced. Therefore, we aim to support the analysis of the reasons behind these outcomes by analysing them separately. Specifically, one can create a rule set that explains the reasons for the TD being either violated or not violated. In some cases, it is also worthwhile to understand which type of TD is generating the error/NA in the VS. Therefore,

Table 12. MRs used and the total number of methods to which a specific MR applies (column '✓')

MR	Change in the input	Output expected	✓	✗
MR_{PER}	Permute the components	Remain constant	26	18
MR_{ADD}	Add a positive constant	Increase or remain constant	36	8
MR_{MUL}	Multiply by a positive constant	Increase or remain constant	37	7
MR_{INV}	Take the inverse of each element	Decrease or remain constant	31	13
MR_{INC}	Add a new element	Increase or remain constant	35	9
MR_{EXC}	Remove an element	Decrease or remain constant	35	9
Total number of cases to which the set of MRs applies (✓) and does not apply (✗)			115	35

when necessary, we also analyse those cases.

Step 3.2.3 - Rule mining: This step is responsible for generating the set of rules by using the Apriori ARM algorithm.

6.2.2. SUT and Predefined Set of MRs

To evaluate our association-rule-based approach we used 44 Python methods from the dataset created by in chapter one. We retained most of the MRs and their labels from our previous work [87], with some label modifications based on our results and experience. Table 12 summarises the MRs used, input and output changes, and the number of methods to which each MR applies. Table 13 lists the methods with their respective labels as reported in [29] and [35]. The column Ground Truth **(GT)** contains the labels, where '1' indicates that the MR always applies, and '0' indicates that it does not always apply. The original labels were assigned under the assumption that all input data are positive.

We divided the methods into three groups, shown in Table 13 under the **GP** columns. Each group was determined based on similarities in the number and types of inputs and the type of outputs among the methods. Group one, **G01**, includes 23 methods that take a `list` as input and produce `Int` or `Float` outputs. Group two, **G02**, consists of 13 methods that take two input `lists` and produce `Int` or `Float` outputs. Group three, **G03**, includes 8 methods that take one `list` as input and produce `list` outputs.

6.2.3. Evaluation and Results

Figure 13 shows the pipeline that implements the MetaTrimmer steps, including the ARM-based module in MR Analysis. The implementation of all steps of the MetaTrimmer pipeline consists of nine Python scripts. The first script, named `InputGenerator.py` (2.1), is a fuzzer that uses a random number generator to create and store in a JSON file the TD. This data generation process relies on a predefined set of parameters, which include:

- low (`l`): Fuzzer minimum value threshold. The minimum number that the fuzzer will consider. For instance, if "low" is set to -1, the fuzzer will not generate elements less than -1.

- high (`h`): Fuzzer maximum value threshold. The maximum number the fuzzer will consider. For instance, if “high" is set to 5, the fuzzer will not generate elements greater than 5.
- input_type (`it`): The type of input.
- t_{end} (`t`): The end time of the program execution.
- output (`o`): The name of the output file.

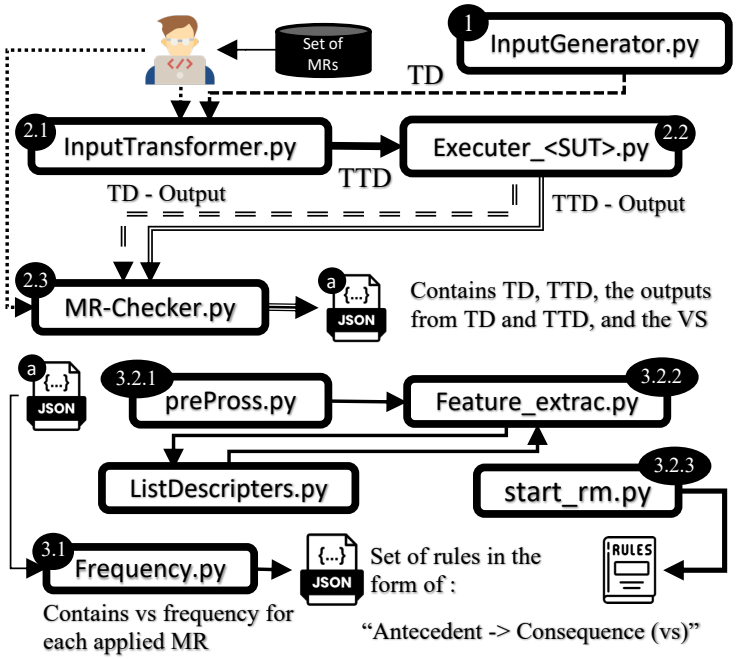

Figure 13. Possible implementation pipeline of MetaTrimmer

The second script, `InputTransformer.py` (2.1), transforms the TD generated by `InputGenerator.py` based on the specifications of the MR. It is important to note that the MRs are described in natural language and need to be translated into code so that they can be applied to the TD. The MRs are hardcoded in `InputTransformer.py` script so that they are applied automatically to the generated TD to generate the transformed TD. When the SUT has two inputs, the transformation is done in both inputs, and is stored in separate subkey of the `JSON` file. The main purpose of the third script, `Executer_<method>.py` (2.2), is to execute the TD and TTD against the SUT. It is important to note that each SUT, there exists an `Executer_<method>.py`. The fourth script, `MR-Checker_n.py` (2.3), consumes the outputs generated by the executer scripts, *i.e.,* `Output-TD` and `Output-TTD`, and verifies whether both outputs match their respective expected change given by the MR.

In our evaluation, there are two types of `MR-Checker.py` (2.3). The distinction arises from differences in output types.

Similar to the InputTransformer.py script, the expected output relations for each MR are hard-coded into the MR-Checker_.py script. This means the script automatically applies these relations to the outputs generated by the executer scripts. At this stage, all the TD, TTD, their corresponding outputs, and their MR VS (violated/non-violated) provided by the MR-Checker.py are stored in a JSON (a) file.

The JSON (a) files generated by the MR-Checker are the Frequency.py inputs. The purpose of Frequency.py is to calculate, for each method, the frequency of violations or non-violations of each applied MR, storing this information in JSON (c) format. This frequency data guides the classification of methods to be analysed further by the ARM-Based Module for pattern extraction. Once the methods and MRs for deeper analysis are identified, the prePross.py (3.2.1) script ensures consistency in the nomenclature used for attribute identification and transforms the JSON object into a DataFrame.

Subsequently, Feature_extract.py (3.2.2) converts the TD into a set of descriptive characteristics. To facilitate this process for list type inputs, we have developed the ListDescripters.py Class, which contains potential descriptive features that can be extracted. To inspire the creation of descriptive features for a List type input, we considered the question, *"How can one effectively describe a list?"* Key characteristics often include the list's size, whether it is empty, and the presence of zeros. Based on these considerations, we compiled a list of 23 descriptors. For our evaluation, we decided to focus on 7 specific descriptors. The complete list is available in our repository. We chose to prioritise the descriptor that are in the form of categorical variables in our initial experiments. However, we recognise that combining categorical variables with other descriptors could produce interesting results. Below, we describe the descriptors we selected for our analysis:

- **size**: Length of the list.
- **isEmpty**: Indicates whether the list is empty or not.
- **has_zeros**: Tells if the list contains any zero values.
- **cnt_zeros**: Counts the number of zero values in the list.
- **cnt_positive_numbers**: Counts the number of positive numbers in the list.
- **cnt_negative_numbers**: Counts the number of negative numbers in the list.
- **get_minimum**: Tells the minimum value from the list.
- **get_maximum**: Tells the maximum value from the list.
- **get_sum**: Indicates the sum of all elements in the list.
- **get_mean**: Indicates the mean (average) value of the elements in the list.
- **get_median**: Retrieves the median value from the list.
- **get_range**: Tells the range (difference between maximum and minimum) of the list.
- **has_duplicates**: Checks if the list contains any duplicate values.

- **is_sorted_asc**: Indicates whether the list is sorted in ascending order.
- **count_distinct_values**: Counts the number of distinct (unique) values in the list.
- **contains_even_numbers**: Checks if the list contains any even numbers.
- **contains_odd_numbers**: Checks if the list contains any odd numbers.
- **contains_integers**: Checks if the list contains any integers.
- **contains_floats**: Checks if the list contains any floating-point numbers.
- **pp_numbers**: Average of positive numbers.
- **pn_numbers**: Average of negative numbers.
- **more_pos_neg**: Determines whether the list contains more positive or negative numbers. it returns “p", indicating that the list contains more positive numbers than negative numbers. It returns “n" when the list contains more negative elements. If both are equal, it returns “e"
- **weight_of_list**: It tells whether the sum of positive numbers in the list is greater than the sum of negative numbers, or vice versa. If the sum of positive numbers is greater, it returns “wp", indicating that the positive numbers have more “weight" or “influence" within the list. If the sum of positive numbers is less, it returns “wn". If both sums are equal, it returns “we".

Once the dataset has been created in the form of descriptors, it is then passed to `start_rm.py (3.2.3)`. We employ the Python3 `mlxtend` library, specifically the `apriori` and `association_rules` functions from the `frequent_patterns` module.

Figure 14 shows the possible values of GT and its corresponding meaning for MetaTrimmer. When GT is set to ‘1’, the MR always applies. On the other hand, a value of ‘0’ means that the MR does not always apply. The symbol ✓ denotes the percentage of runs when the MR applies, and it has two possible outcomes: If ✓ is 100%, we say that the GT is confirmed by MetaTrimmer. These cases are marked in Table 13 using non-bold and non-italics formatting. However, if it is less than 100%, we assume the GT is contradicted. In such cases, we distinguish between two types of contradiction: if ✓ is equal to 0, it means that the GT is fully contradicted. If ✓ is less than 100% but not equal to 0, we refer to it as a mixed case, where the GT is partially contradicted. These cases are marked in Table 13 using bold formatting. Similarly, the symbol ✗ denotes the percentage of runs where the MR does not apply or is violated. If ✗ is 100%, we say that the GT is fully confirmed. These cases are marked in Table 13 using non-bold and non-italics formatting. However, if it is less than 100%, we may consider it a mixed case, where the GT could be partially confirmed. These cases are marked in Table 13 using bold formatting. If ✗ is equal to 0, it means that the GT is contradicted. Since the GT was defined under the assumption that only positive input data is used, but our experiments allow both positive and negative input data, contradictions to the GT are to be expected.

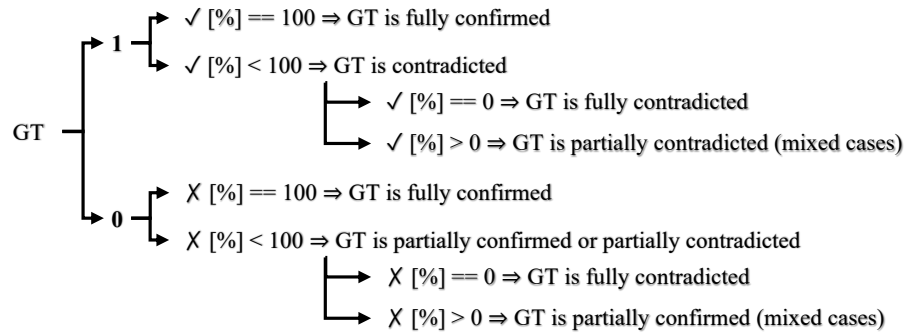

Figure 14. GT values for MetaTrimmer evaluation: 1 means the MR always applies, while 0 means it doesn't always apply. We use symbols ✓and ✗ to indicate the percentage of runs where the MR applies or is violated. ✓at 100% means the GT is confirmed. Otherwise, it is contradicted. If ✓ is 0, GT is fully contradicted; if it's less than 100% but not 0, it's partially contradicted (mixed case). Similarly, ✗ at 100% means the GT is fully confirmed. Otherwise, it could be partially confirmed (mixed case), but if ✗ is 0, the GT is contradicted.

Upon initial examination of the results presented in Table 13, one can observe three types of outcomes: violation, non-violation, and invalid data. When analysing the frequency results presented, one can identify at least three patterns. *i)* It is evident that the applicability of MRs can be easily determined by checking if there are 100% no violations, *ii)* it is also evident that the non-applicability of MRs can be easily detected, *iii)* the mixed cases are mostly in the group **G01**. In terms of classification based on the MetaTrimmer statement, a consistent 100% violation rate indicates that the MR does not apply to the SUT. Conversely, a non-violation rate suggests that the MR aligns with the SUT's behaviour. MetaTrimmer achieves a compliance rate of 100% for all methods labelled as '1' in the GT. This means that for methods correctly identified as compliant with the SUT, MetaTrimmer accurately recognises this alignment.

For the constraint definition analysis using the proposed ARM-based approach, we focused on the mixed cases, which are marked in bold formatting in Table 13. Table 14 provides an example of the rule set generated for the method `add_values` (**id**: 1). As shown in Table 13, this method exhibits a 52% non-violation rate, while violations account for 48% for the MR_{MUL}. As Table 14 shows, the rule set consists of Antecedents and a Consequent. When analysing the instances marked as violations, we consider only the rules where the Consequent is solely "Violation", we exclude any rules where the Consequent combines "Violation" with other attributes, as we are not interested in those combinations. Similarly, we apply this filtering when analysing non-violation cases and error/NA instances when necessary. In addition, we sorted the rules in descending order based on their confidence levels. As we mentioned in Section 3.4, confidence measures the likelihood that the Consequent is true given the Antecedents. By

Table 13. Set of methods with the GT from [29] and [35]: '1' means that the MR always applies, and '0' means that the MR does not always apply. Symbol ✓ denotes the percentage of runs when the MR applies, and symbol ✗ denotes the percentage of runs when the MR does not apply (= is violated). Symbol − denotes the percentage of errors/invalid data.

GP	id	Method name	MR_{PER}				MR_{ADD}				MR_{MUL}				MR_{INV}				MR_{INC}				MR_{EXC}			
			GT	✓ [%]	✗ [%]	− [%]	GT	✓ [%]	✗ [%]	− [%]	GT	✓ [%]	✗ [%]	− [%]	GT	✓ [%]	✗ [%]	− [%]	GT	✓ [%]	✗ [%]	− [%]	GT	✓ [%]	✗ [%]	− [%]
G01	1	add_values	1	100	0	0	1	100	0	0	**1**	**52**	**48**	0	**1**	**44**	**42**	**14**	1	100	0	0	**1**	**53**	**47**	0
	2	average	1	93	0	7	1	93	0	7	**1**	**45**	**48**	**7**	**1**	**37**	**42**	**21**	**0**	**61**	**32**	**7**	**0**	**43**	**45**	**13**
	3	cnt_non_zeros	1	100	0	0	1	87	13	0	1	100	0	0	1	86	14	0	1	100	0	0	1	100	0	0
	4	cnt_zeros	1	100	0	0	0	88	12	0	0	100	0	0	0	100	0	0	1	100	0	0	1	100	0	0
	5	durbinWatson	0	13	80	7	**0**	**47**	**46**	**7**	1	93	0	7	**1**	**41**	**38**	**21**	1	**58**	**35**	**7**	**1**	**42**	**45**	**13**
	6	entropy	1	13	0	87	1	8	4	88	1	13	0	87	0	11	1	88	1	6	0	94	1	7	0	93
	7	find_magnitude	1	93	0	7	**1**	**49**	**44**	**7**	1	93	0	7	1	79	0	21	1	93	0	7	1	87	0	13
	8	find_max2	0	25	63	13	1	87	0	13	1	76	11	13	1	63	11	26	1	82	5	13	1	75	6	19
	9	find_max	1	93	0	7	1	93	0	7	1	86	7	7	1	73	7	21	1	93	0	7	1	87	0	13
	10	find_median	1	93	0	7	1	93	0	7	**1**	**45**	**48**	**7**	**1**	**37**	**43**	**21**	0	60	33	7	**0**	**47**	**40**	**13**
	11	find_min	1	93	0	7	1	93	0	7	1	6	87	7	1	6	73	21	0	88	5	7	0	73	14	13
	12	geometric_mean	1	5	0	95	1	5	0	95	1	5	0	95	1	5	0	95	0	0	5	95	0	2	1	97
	13	harmonicMean	1	5	0	95	1	5	0	95	1	5	0	95	1	5	0	95	0	0	5	95	0	2	1	97
	14	kurtosis	1	76	0	24	1	76	0	24	1	76	0	24	0	8	54	38	0	60	16	24	0	42	27	31
	15	max	1	93	0	7	1	93	0	7	1	86	7	7	1	73	7	21	1	93	0	7	1	87	0	13
	16	min	1	93	0	7	1	93	0	7	1	6	87	7	0	6	73	21	0	88	5	7	0	73	14	13
	17	product	1	93	0	7	**1**	**53**	**40**	**7**	**1**	**54**	**39**	**7**	**1**	**40**	**39**	**21**	**1**	**54**	**39**	**7**	**1**	**51**	**37**	**13**
	18	safeNorm	1	93	0	7	**1**	**49**	**44**	**7**	1	93	0	7	1	79	0	21	1	93	0	7	1	87	0	13
	19	sampleVariance	1	87	0	13	1	87	0	13	1	87	0	13	1	74	0	26	0	9	79	13	0	33	48	19
	20	skew	1	81	0	19	1	81	0	19	1	81	0	19	**0**	**33**	**35**	**32**	0	**28**	**53**	**19**	**0**	**38**	**37**	**24**
	21	**sumOfLogarithms**	**1**	**12**	**0**	**88**	**1**	**12**	**0**	**88**	**1**	**12**	**0**	**88**	**1**	**12**	**0**	**88**	**1**	**12**	**0**	**88**	**1**	**12**	**0**	**88**
	22	sum	1	100	0	0	1	100	0	0	**1**	**52**	**48**	**0**	**1**	**44**	**42**	**14**	1	100	0	0	**1**	**54**	**46**	0
	23	variance	1	93	0	7	1	93	0	7	1	93	0	7	1	79	0	21	0	17	76	7	**0**	**43**	**44**	**13**
G02	24	add	0	16	84	0	1	100	0	0	1	14	86	0	1	13	43	44	1	100	0	0	1	94	6	0
	25	calculateDifferences	0	16	84	0	0	6	94	0	0	12	88	0	0	11	45	44	1	100	0	0	1	94	6	0
	26	dec	0	16	84	0	0	6	94	0	0	12	88	0	0	11	45	44	1	100	0	0	1	94	6	0
	27	ebeAdd	0	16	84	0	1	100	0	0	1	14	86	0	1	13	43	44	1	100	0	0	1	94	6	0
	28	ebeDivide	0	15	58	27	0	12	44	44	0	6	67	27	0	11	45	44	1	73	0	27	1	68	8	23
	29	ebeMultiply	0	16	84	0	1	21	79	0	1	12	88	0	1	12	44	44	1	100	0	0	1	94	6	0
	30	ebeSubtract	0	16	84	0	0	6	94	0	0	12	88	0	0	11	45	44	1	100	0	0	1	94	6	0
	31	elementwise_equal	0	72	28	0	0	6	94	0	0	6	94	0	0	6	94	0	1	100	0	0	1	94	6	0
	32	elementwise_max	0	23	77	0	1	100	0	0	1	25	75	0	1	19	37	44	1	100	0	0	1	94	6	0
	33	elementwise_min	0	23	77	0	1	100	0	0	1	9	91	0	1	8	47	44	1	100	0	0	1	94	6	0
	34	elementwise_not_equal	0	72	28	0	0	6	94	0	0	6	94	0	0	6	94	0	1	100	0	0	1	94	6	0
	35	find_diff	0	16	84	0	0	6	94	0	0	12	88	0	0	11	45	44	1	100	0	0	1	94	6	0
	36	pooledMean	1	94	0	6	1	94	0	6	**1**	**41**	**53**	**6**	**1**	**23**	**27**	**50**	1	83	11	6	1	44	44	12
G03	37	array_copy	0	0	100	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	38	bubble	0	0	100	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	39	insertion_sort	1	100	0	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	40	reverse	1	100	0	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	41	selection_sort	1	100	0	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	42	shell_sort	1	100	0	0	1	100	0	0	1	88	12	0	1	12	74	14	1	100	0	0	1	93	7	0
	43	square	0	100	0	0	1	13	87	0	1	86	14	0	1	65	21	14	1	100	0	0	1	93	7	0
	44	standardize	0	72	15	13	1	1	87	13	1	87	0	13	0	0	73	27	1	87	6	7	1	81	6	13

GP: Group, **GT:** Ground Truth

sorting rules by confidence in descending order, we prioritise the most reliable and strong associations, ensuring that the most trustworthy rules are examined first.

From the rule set that has "Violation" as the consequent in Table 14, it is evident that the attribute **weight_of_list_wn** is a strong indicator for triggering the violation. This attribute not only ranks at the top of the rule set but also appears in combination with other attributes in the top five rules, emphasising its significance. When we examine the rule set for "Non-violation," **weight_of_list_wn** is notably absent from the antecedents, further confirming its role as a strong pattern associated with violations. Conversely, in the rule set for "Non-violation," the attribute **weight_of_list_wp** emerges as a strong indicator. It ranks at the top among the antecedents and appears in combination with various other attributes, reinforcing its significance in predicting non-violations. This contrast between the presence of **weight_of_list_wn** in violation rules and **weight_of_list_wp** in non-violation rules highlights the distinct patterns that these attributes contribute to in the analysis. Based on the rule set, one can conclude that when applying

the MR_{MUL} to the method `add_values`, it is necessary to consider that if the sum of the positive elements is lower than the negative elements, the violation will be triggered. To avoid this, the MR_{MUL} could be refined by constraining the TD space as follows:

> **MR_{MUL}:** multiply each TD input element with a positive constant $k > 1$, *i.e.*,
>
> $$TTD_n = \{x_1 \times k,\ x_2 \times k,\ ...,\ x_n \times k\}$$
>
> Then the following output-relation must hold:
>
> $$Output_{TTD} \geq Output_{TD}$$

Refinement:

> **if:**
>
> $$\sum_{i=1}^{n} x_i^+ \geq \left|\sum_{i=1}^{n} x_i^-\right|$$
>
> where X_i^+ represents the positive elements of the list and X_i^- represents the negative elements of the list. Then the following output-relation must hold:
>
> $$Output_{TTD} \geq Output_{TD}$$
>
> **else:**
>
> $$Output_{TTD} < Output_{TD}$$

While Table 14 showcases a successful and straightforward example of our approach, Table 15 shows a not straightforward outcome. Table 15 presents the rule set generated for the method `durbinWatson` when analysing MR_{ADD}. The highlighted antecedents indicate contradictions within the rule set. For instance, the rules suggest that a violation occurs when **weight_of_list_wn** is present, but also when **weight_of_list_wp** and **weight_of_list_e** are present. These three attributes are highlighted in blue. In total 10 out of 14 antecedents are contradictions. The contradicting atributes are higligted in orange, yellow, gray and blue in Table 15. In total, 10 out of 14 antecedents are contradictory, showing conflicting patterns that make it difficult to derive a clear understanding of what leads to a violation. This highlights a limitation in the ARM-based approach, as the presence of multiple conflicting rules reduces the reliability of the extracted patterns. Furthermore, regarding the "Non-violation" rule set, the algorithm was not able to generate a single rule with only the consequence "Non-violated." This suggests that the algorithm struggled to find consistent patterns that exclusively lead to non-violations, further complicating the analysis. This inability to generate non-violation rules indicates that the method `durbinWatson` does not exhibit clear, consistent behaviour under MR_{ADD}, pointing to potential issues either in the method's implementation or in the suitability of the MR for this method.

Table 14. Rule set generated for the method `add_values` when analysing MR_{MUL}.

Antecedents	Consequent	sup	conf	lift	ZM
[**weigth_of_list_wn**]	Violation	0.38	1.00	2.63	1.00
[**weigth_of_list_wn**, has_zeros_True]		0.17	1.00	2.63	0.74
[**weigth_of_list_wn**, has_duplicates_True]		0.17	1.00	2.63	0.74
[*more_pos_neg_n*, **weigth_of_list_wn**]		0.17	1.00	2.63	0.74
[**weigth_of_list_wn**, *more_pos_neg_e*]		0.12	1.00	2.63	0.70
[*more_pos_neg_n*]		0.17	0.50	1.31	0.36
[has_zeros_True]		0.17	0.44	1.15	0.21
[has_duplicates_True]		0.17	0.44	1.15	0.21
[more_pos_neg_e]		0.12	0.36	0.94	-0.09
[**weigth_of_list_wp**]	Non-violation	0.43	1.00	1.62	0.67
[has_zeros_True, **weigth_of_list_wp**]		0.19	1.00	1.62	0.47
[has_duplicates_True, **weigth_of_list_wp**]		0.17	1.00	1.62	0.46
[**weigth_of_list_wp**, more_pos_neg_p]		0.17	1.00	1.62	0.46
[is_sorted_True, **weigth_of_list_wp**]		0.14	1.00	1.62	0.44
[more_pos_neg_e, **weigth_of_list_wp**]		0.14	1.00	1.62	0.44
[more_pos_neg_n, **weigth_of_list_wp**]		0.12	1.00	1.62	0.43

sup: support, **conf:** confidence, **ZM:** zhangs_metric

The method `durbinWatson` is used to detect the presence of auto correlation in the residuals of a regression analysis. It is calculated using the formula:

$$d = \frac{\sum_{t=2}^{n}(e_t - e_{t-1})^2}{\sum_{t=1}^{n} e_t^2}$$

where e_t are the residuals (errors) from the regression model. The differences between residuals are calculated as follows:

$$e_2 - e_1 = r_1, e_3 - e_2 = r_2, e_4 - e_3 = r_3$$

where r represents the differences between consecutive residuals. From the domain knowledge, we understand that this methods creates a complex dependency structure that is sensitive to the values of all preceding and following residuals. As a result, small changes in the input data can lead to significant variations in the output, making it difficult to identify a consistent pattern. The inability to generate a rule set with non-violation as the sole consequence indicates that the method does not consistently behave in a predictable manner under MR_{ADD}. This lack of clear, consistent behaviour suggests that the underlying MR may not be well-suited. This conclusion aligns with the **GT**.

The full set of rules for each method and its respective mixed case MR outcomes can be found in our GitHub repository. Overall, the top antecedents in

Table 15. Rule set generated for the method `durbinWatson` when analysing MR_{ADD}. The colours indicate contradictions within the rule set: blue highlights attributes such as weight_of_list_wn that contradict with other antecedents like weight_of_list_wp and weight_of_list_we; orange, yellow, and grey represent additional contradictions.

Antecedents	Consequent	sup	conf	lift	ZM
[has_zeros_True, weight_of_list_wp]	Violation	0.15	1.00	1.57	0.43
[has_duplicates_True, weight_of_list_wp]		0.13	1.00	1.57	0.42
[more_pos_neg_e, weight_of_list_wp]		0.11	1.00	1.57	0.41
[weight_of_list_wp]		0.33	0.95	1.49	0.50
[more_pos_neg_p, weight_of_list_wp]		0.13	0.88	1.38	0.32
[weight_of_list_wp, is_sorted_True]		0.11	0.86	1.35	0.30
[weight_of_list_we]		0.11	0.75	1.18	0.18
[more_pos_neg_p, has_duplicates_True]		0.11	0.75	1.18	0.18
[more_pos_neg_p]		0.22	0.71	1.11	0.14
[more_pos_neg_e]		0.22	0.67	1.05	0.07
[has_zeros_True, has_duplicates_True]		0.11	0.67	1.05	0.05
[more_pos_neg_n]		0.20	0.58	0.91	-0.13
[has_duplicates_True, weight_of_list_wn]		0.11	0.46	0.73	-0.33
[weight_of_list_wn]		0.20	0.39	0.62	-0.56

sup: support, **conf:** confidence, **ZM:** zhangs_metric

rule sets for methods presenting mixed cases in MR_{ADD} and MR_{MUL} were **weight_of_list_wn**, along with **more_pos_neg_n** and **has_zeros_True**. For MR_{INV}, the top antecedent in the rule set that triggered violations was **weight_of_list_wn**, while the antecedents that triggered error/NA were **isEmpty_True** and **has_zeros_True**. Regarding MR_{INC} and MR_{EXC}, our approach did not find any helpful patterns. This outcome was expected since the essence of these MRs involves modifications not within the elements of the list but in an external manner, specifically by altering the size of the list. In this initial proof-of-concept, we focused solely on the relationship between TD and VS without considering the TTD. Consequently, our approach was not aware of changes related to the description of the TD, rendering the rule set suboptimal.

6.2.4. Threats to Validity

In the context of our study, two types of threats to validity are most relevant: threats to internal and external validity.

To achieve internal validity, we used the same workflow of MetaTrimmer as in [87]. There is a potential risk of bias in selecting the libraries of ML or the TD used, which could potentially impact the results obtained.

To enhance external validity and ensure the generalisability of the results, it

would have been preferable to include a wider range of methods in the evaluation process. This would have helped to mitigate any potential bias that could have arisen from the classification of the methods used. Therefore, the current study may not be able to determine the full extent of the effectiveness of the pattern extraction with the help of MetaTrimmer.

6.3. Discussion

In this chapter, we addressed RG_2, which aims to provide a method for refining MRs by setting constraints based on TD. To achieve this, we leveraged ARM to automate part of the MR Analysis step, which previously relied heavily on manual inspection. By applying ARM, we systematically identified patterns and associations between the characteristics of the TD and the MR violation status. These patterns were then used to define constraints, specifying the precise input space where each MR is valid.

Below, we discuss the answers to the research questions based on our findings:

6.3.1. $RQ_{2.1}$: How can patterns be extracted and utilised as constraints to refine MRs?

We addressed this research question by introducing ARM into the MetaTrimmer process. ARM facilitates the extraction of patterns between TD characteristics and MR outcomes. Through this, we moved from manually analysing mixed cases—where MRs exhibit both violations and non-violations—to automatically identifying patterns that define when an MR applies to specific subsets of input data.

For example, as shown in the illustrative example (see Table 11), when the TD consists of positive numbers, the MR is not violated, but when negative numbers are present, violations occur. By using ARM, such patterns can be automatically derived, reducing the need for manual inspection and providing a more systematic approach to defining constraints for MRs.

This process enables us to refine MRs by extracting actionable insights from the mixed cases, revealing under which conditions an MR is valid. These constraints ensure that MRs are applicable to relevant input data, improving the precision of MR classification

6.3.2. $RQ_{2.2}$: How well does the ARM-driven approach refine and constrain MRs?

In addressing $RQ_{2.2}$, the evaluation highlighted the effectiveness of the ARM-driven method in refining MRs through the extraction of patterns from the TD. Out of the 44 Python methods tested, across 6 MRs, 36 mixed cases were identified, where the MRs did not consistently exhibit violations or non-violations.

The ARM approach successfully generated meaningful constraints for 68% of the cases, enabling a clear delineation of when MRs were valid. By transforming TD characteristics into association rules, ARM facilitated the discovery of patterns such as when specific conditions in the TD would trigger violations or non-violations. For instance, in several cases, ARM identified key features in the input data—such as the relative weights of elements in a list—that could strongly predict MR violations, thereby helping refine the MR and reduce false positives.

However, 32% of the cases required deeper manual analysis. In these cases, the ARM-generated rules exhibited contradictions or failed to yield clear, actionable insights. This suggests that while ARM is effective in most scenarios, it struggles when the method or the MR presents complex dependencies or nonlinear relationships within the input data. For such cases, manual inspection and possibly the application of more sophisticated analysis techniques are needed to fully understand and refine the MRs.

6.4. Replication Packages and Artifacts

- Bug or not bug? (related to publication V)
  https://github.com/aduquet/VST2023-BugORNOTbug
- Metatrimer+ (related to publication VIII)
  https://github.com/aduquet/SelectingConstrainingMRsScSoft

7. ASSESSING THE STRENGTHS OF METAMORPHIC RELATIONS

Chapter 7 is based on the publication VI and addresses RG_3 by aiming to define and evaluate an approach that assesses the strength of MRs in terms of their defect-detection capabilities. To answer the research question RQ_3: *What TD-based method can be used to assess the strength of MRs regarding their bug-finding capability?*, we provide a three-phase approach, which combines mutation testing with MetaTrimmer. First, we evaluate the applicability of each MR using MetaTrimmer. Then, we perform mutation testing. Finally, we employ a three-level strategy to evaluate MR effectiveness: level one focuses on their ability to detect equivalent mutants; level two triggers violations when mutants are killed across the entire input space; and level three involves sensitivity analysis. We use a black-box industrial optimisation algorithm as the SUT to evaluate our approach.

In addition, this chapter aims to describe our observations and findings from applying the MT approach in an industrial context, including insights into successful aspects and areas that presented challenges. Additionally, we identify gaps in current MT research, highlighting opportunities for further exploration and improvement. We also share valuable feedback received from our industry partner. By applying the MT approach to a real-world industrial optimisation algorithm, our research aims to contribute to the advancement of testing methodologies for complex algorithms. The integration of MetaTrimmer with mutation testing, along with our three-level strategy, offers a promising approach for evaluating and ranking chosen MRs in terms of their defect-detection effectiveness.

7.1. Testing Optimisation Algorithms

Optimisation Algorithms (OAs) are search methods designed to find a solution to an optimisation problem, aiming to optimise a given quantity, potentially subject to a set of constraints. Through systematic exploration, these algorithms traverse solution spaces to identify the most beneficial or nearly optimal solutions for the given problem [88]. Their main goal, therefore, is to identify the most favourable values for a given set of decision variables, whether it entails maximising or minimising a specific objective function [89]. By doing so, OAs play an important role in tackling real-world challenges and refining decision-making processes across various fields. For instance, in machine learning, several OAs, such as grid search [90], random search [91], Bayesian optimisation [92] among others, had a significant role in parameter tuning, resulting in major improvements in the performance of a wide range of ML algorithms [93].

As the practical applications of OAs continue to expand across diverse domains, evaluating and ensuring their quality becomes crucial. However, assessing

the correctness of OAs is not a trivial task. In many real-world optimisation problems, there may be no known optimal solution or ground truth against which to evaluate the output of a particular OA. Without a ground truth solution, it becomes challenging to determine whether an algorithm has converged to the best possible outcome or if there is still room for improvement. Therefore, OAs often fall into the category of *"non-testable"* programs, where a reliable oracle demands a significant investment of time and resources [94]. It is important to note that in the context of OAs, if the oracle were know, there have been no need of them.

In our research [10] (publication VI), we explored the practical application of MT on a black-box industrial optimisation algorithm used in the field of machinery and plant engineering. Our focus was on identifying and implementing MRs that reflected the algorithm's behaviour. We also shared the insights gained throughout this process, including feedback and reception from our industry partner regarding the MT approach. Building on this work, we conducted a comprehensive study to assess the effectiveness of the previously identified Handcrafted MRs (HMRs) in terms of their bug-finding capabilities, and complementing with Generic MRs (GMRs). We refer to GMRs as a set of MRs that are derived from 'universal or generic' properties. The goal of this study was to provide an answer to *RQ_3: What TD-based method can be used to assess the strength of MRs regarding their bug-finding capability?*

In response to our question, we introduce a three-phase approach that integrates MetaTrimmer with traditional mutation testing. Our approach begins by evaluating the applicability of each MR based on TD using MetaTrimmer (Phase 1 - Applying MetaTrimmer). Subsequently, we perform mutation testing (Phase 2 - Mutation Testing), which involves generating mutated versions of the SUT, verifying their executability, and conducting initial analyses to determine the fate of mutants (killed, survived, or equivalent). Following this, we leverage MetaTrimmer with the mutated SUT versions and compare them with the rates of killed versus survived from the mutation testing phase (Phase 3 - Assessing the Effectiveness of HMRs/GMRs). To ensure a comprehensive assessment of MR effectiveness, we employ a three-level strategy to compare outcomes with those obtained from mutation testing. The first level aims to assess the MRs' capability to detect equivalent mutants. In the second level, we evaluate whether the MRs can trigger violations when a particular mutant is killed during mutation testing across all TD input. Lastly, the third level focuses on assessing the sensitivity of the MR. Here, sensitivity is defined by the amount of TD required to elicit a violation. A higher sensitivity is indicated when a MR triggers a violation for the same mutant with a larger TD input set.

The integration of MetaTrimmer with mutation testing, along with our three-level strategy, sets our approach apart from others works. Through our approach, we have demonstrated that when mutants are consistently eliminated by all TD, as analysed through traditional mutation testing, an MR is able to detect these mutants by triggering violations in at least one TD input. Additionally, we estab-

lish that an effective MR requires high sensitivity to be truly useful. Our study highlights scenarios in which certain TD fail to detect mutants through traditional mutation testing, yet MRs successfully uncover them. A highly sensitive MR operates effectively across a broader range of inputs, making it especially valuable in industrial contexts such as our SUT.

7.2. Industry Context and System Under Test

This section provides a deeper understanding of the environment within which the SUT operates. Starting with Section 7.2.1, we delve into the specific industry context, offering insights into the broader context that influences the SUT's operations. In Section 7.2.2, although the details of the SUT remain confidential as per our agreement with our industrial partner, we aim to offer insight into the functionality and operational aspects of the SUT while ensuring proprietary information is protected. In Section 7.2.3, we shed light on how we extracted the HMRs.

7.2.1. Industry Context

The work has been performed on an industrial use case in the field of machinery and plant engineering. The machinery of the company can produce and handle different products for different domains depending on the needs of the customer. A huge number of different machine variants and configuration options are available, which specify the size, power, features, and versatility of its different product lines. These machines are part of a larger production process, and their exact use is unknown to the manufacturer. Therefore, there is the need to configure and optimise the machines and the production process by the customer. The ability of the machine to easily change for different purposes is an important feature for customers. This makes these machines versatile, and they can be converted for the production of other items in a short time. Although a working machine configuration can be saved and reused later on, there is still room for improvement in the partly manually determined settings. Besides changes in environmental conditions, like temperature, which might require adapting some settings of the production process, there are additional reasons to do so. Sustainability is supported by trying to reduce the energy and resource consumption of the production process without compromising the quality of the resulting products.

The machine vendor incorporates code that supports half-automatic optimisation of parts of the production process. Its goal is to decrease production cycle time and reduce machine resource consumption without compromising product quality. After the machine's initial setup, *i.e.,* once it can produce with the required quality, the operator can initiate the optimisation process. It seeks to refine the production process within the given frame by running cycles and adjusting values that influence the cycle. These settings can be adopted for subsequent use if better values are identified.

It is obvious that supporting the whole range of different machines, machine configurations, and options is a challenge. Especially if there is the need to adapt the algorithm to support specific equipment of a machine, it has to be ensured that the changes do not have any side effects on other plant configurations and still provide valid results. Even the optimisation code has to be reliable, *e.g.,* when replacing sensors. The optimisation code is currently written as MATLAB scripts and generated to control code during the software build process. Most of the testing activities of the optimisation features are performed manually since it is part of a complex production cycle, which is difficult to automate and realistically simulate currently. In addition to these integration tests, there already existed manually created test scripts in MATLAB that have been used to test some important parts and single functions of the optimisation process.

We could convince our long-lasting company partner to research the potential of MT in the context of this optimisation code. The optimisation algorithm follows some rules which must hold true for all machine types, which, therefore, seemed to be the perfect use case for MT. With the application of MT, we wanted to increase the coverage of some selected central functions of this code and show that the defined assumptions are valid for the specified input space.

7.2.2. System Under Test

The SUT is an optimisation algorithm implemented in MATLAB. It is designed to search for the best fitting of two dependent lines using the least squares method and determine their intersection point. This algorithm is part of a larger system aimed at refining the production process by decreasing production cycle time and reduce machine resource consumption without compromising product quality. While we are constrained by confidentiality agreements with our esteemed company partner, we can shed light on the functionality of the SUT at a high level.

The SUT can be explained at four breaking points, each representing a distinct aspect or stage of its functionality and operation:

1) Input data processing:
    - The SUT takes two sets of input data, X_{data} and Y_{data}, representing the x and y coordinates of data points, respectively.
    - It sorts the input data points based on their x-coordinates to prepare for subsequent calculations. This ensures that the data points are arranged sequentially along the $x-$axis.
2) Segmentation and Line Fitting:
    - The algorithm employs the least squares method to fit two dependent linear segments, denoted as L_1 and L_2, through the sorted input data points. This is achieved by minimising the sum of squared residuals between the observed data points and the values predicted by the linear models.

- The linear equations for the segments are defined as follows, with x_i denoting the intersection point on the x-axis:

$$L_1 : y = a_1 + b_1 \cdot x, \quad \text{for} \quad x \leq x_i \tag{7.1}$$

$$L_2 : y = a_2 + b_2 \cdot x, \quad \text{for} \quad x > x_i \tag{7.2}$$

- The segmentation process is iterative, with the algorithm exploring different potential breaking points (x_i) within the sorted data points to determine the optimal division for fitting L_1 and L_2. Each iteration involves:
  - Selecting a segment of data points up to a certain index (ii) to fit L_1 and the remainder to fit L_2.
  - Adjusting the segment boundary (x_i) to minimise the fitting error, quantified as the sum of squared residuals for both lines across all designated points.
- Once the algorithm identifies the optimal breakpoint, it has effectively segmented the data points. This segmentation is where one line will fit the data points before the breakpoint, and the other line will fit the data points after the breakpoint.

3) Calculation of the intersection point (x_{int}, y_{int}) of L_1 and L_2:
   - It determines the $x-$coordinate of the intersection point by solving the equation $a_1 + b_1 * x = a_2 + b_2 * x$
   - A check is performed to ascertain if the lines have different slopes $(b_1 \neq b_2)$, indicating the existence of an intersection point.

4) Outputs generated:
   - x_{int}: This output represents the x-coordinate of the intersection point between lines L_1 and L_2.
   - y_{int}: This output represents the y-coordinate of the intersection point between lines L_1 and L_2. It indicates the value of y at the intersection, which is the common output for the given x_{int}.
   - PX_{data}: This output is a vector that provides the x-coordinates of the lines L_1 and L_2.
   - PY_{data}: This output is a vector that provides the y-coordinates of the lines L_1 and L_2.

Figure 15 provides a visual representation of the outputs of the SUT given certain inputs. The data points are denoted by orange crosses (X_{data}, Y_{data}). Two distinct linear regression lines, L_1 and L_2, are fitted to these points and depicted as solid blue lines. These lines converge at a critical juncture, indicated by a pink dot, annotated as (X_{int}, Y_{int}), marking the break-point between the two fitted lines. The slope of the first line, L_1, is noticeably steeper, suggesting a more pronounced

decreasing trend in the initial segment of the data points. In contrast, the second line, L_2, has a gentler slope, implying a subtler decline in the latter segment. Implicit in the fitting process is the minimisation of an error criterion, *i.e.,* the sum of squared errors, to ensure the lines closely align with the data points.

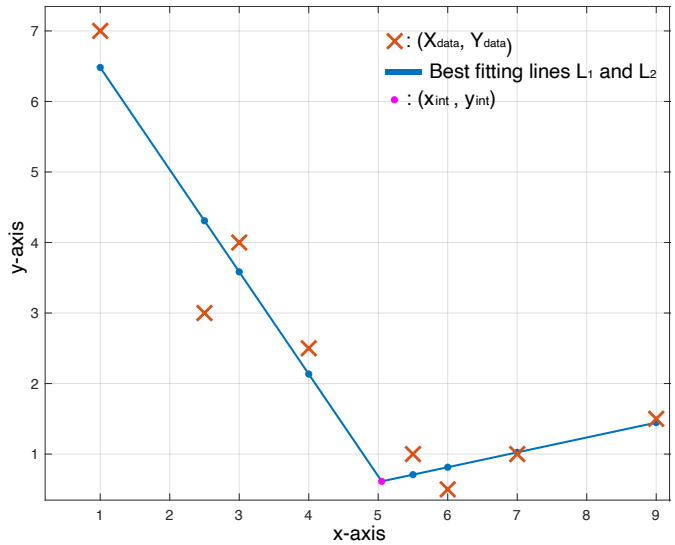

Figure 15. Graphical representation of the SUT inputs and outputs

7.2.3. Formulation of Handcrafted Metamorphic Relations

The process of identifying suitable HMRs involved a series of collaborative workshops with our industrial partner. These sessions started with a detailed introduction to the SUT, proceeded with a thorough presentation of MT approach, and showcased examples of MRs from the research perspective as well as industry projects. The discussions then led to the proposal of potential HMRs. As a result of these joint efforts, four MRs were put forward.

The HMRs were structured in an 'IF-THEN' format, embodying a logical implication where both the antecedent ('IF' part) and the consequent ('THEN' part) constitute relations defined over the inputs and outputs of the function. Thus, each HMR consists of two essential elements: the condition that activates the relation (the 'IF' component) and the anticipated alteration in the output upon fulfilling this condition (the 'THEN' component). We focus on three of the four HMRs initially proposed, as these were deemed most appropriate for the SUT [10]. Detailed descriptions of these HMRs are presented in Section 7.3.1.

7.3. Methodology

In this section, we provide a detailed description of the HMRs in Section 7.3.1, and the GMRs in Section 7.3.2. Additionally, in Section 7.3.3, we outline the methodology adopted for conducting our experiments, which are designed to address the research questions.

7.3.1. Handcrafted Metamorphic Relations

As mentioned in Section 7.2.3, we identified four HMRs, which were identified based on domain knowledge of experts. Drawing upon the findings from our previous work [10], we determined that, for a valid pair of input coordinates (x and y), three out of the four HMRs are indeed applicable to the SUT. Therefore, this study focuses only on those HMRs that are applicable to the SUT based on the result of or previous work [10]. Below a detailed descriptions of the HMRs are provided.

HMR_{1-SHT} - Shift of the Input Data Points. Let $X_{data} = x_1, x_2, ..., x_n$ and $Y_{data} = y_1, y_2, ..., y_n$ represent the original input data points. Let (x_{int}, y_{int}) be the intersection point of lines L_1 and L_2 calculated by the SUT. $\mathbf{v} = (v_x, v_y)$ is a 2D vector representing the shift in the x and y coordinates, respectively.

IF all input data points in X_{data} and Y_{data} are shifted by a certain vector $\mathbf{v}$, *i.e.,* $X_{shifted} = \{x_i + \mathbf{v}_x | \text{ for all } i\}$, and $Y_{shifted} = \{y_i + \mathbf{v}_y | \text{ for all } i\}$

THEN the resulting intersection point of lines L_1' and L_2' should also be shifted by the same vector $\mathbf{v}$, this is:

$$(x_{int} + \mathbf{v}_x,\ y_{int} + \mathbf{v}_y) = IntersectionPoint(SUT(X_{shifted}, Y_{shifted}))$$

Figure 16 illustrates the element-wise shift of the input data by a 2D vector $\mathbf{v} = (v_x, v_y)$, representing the shifts in the x and y coordinates, respectively, as explored in our previous work [10]. The original input data points, marked with orange X's (X_{data}, Y_{data}), contrast with the green X's, which denote the data post-shift. HMR_1 evaluates whether the newly computed lines L_1 and L_2, along with their intersection point, align with the shift imparted by vector $\mathbf{v}$. Through Figure 16, we visually demonstrate MR_1's underlying principle, assessing the shifted input data against the expected spatial translation to ensure the output's fidelity to the applied vector transformation.

HMR_{2-PER} - Permutation of the input data points. Let $X_{data} = x_1, x_2, ..., x_n$ and $Y_{data} = y_1, y_2, ..., y_n$ represent the original input data points.

Let (x_{int}, y_{int}) be the intersection point of lines L_1 and L_2 calculated by the SUT, *i.e.,* IntersecPoint$(SUT(X_{data}, Y_{data}))$.

Let $X_{PER} = x_3, x_n, ..., x_1$ and $Y_{PER} = y_3, y_n, ..., y_1$ represent the shuffled input points obtained by randomly permuting the elements of (X_{data}, Y_{data}).

Let (x_{intP}, y_{intP}) be the intersection point of lines L_1' and L_2' calculated by the SUT, *i.e.,* IntersecPoint$(SUT(X_{PER}, Y_{PER}))$.

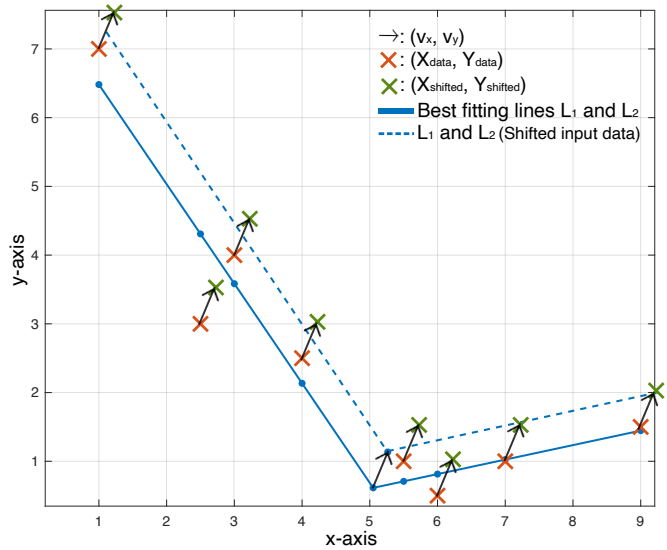

Figure 16. Graphical representation of HMR_{1-SHT} - Shift of the Input Data

IF the data points (X_{data}, Y_{data}) are permuted to create a new input data points (X_{PER}, Y_{PER}),

THEN using the permuted input data points (X_{PER}, Y_{PER}) should produce the same results as the original data point (X_{data}, Y_{data}). this is:

$$\text{IntersecPoint}(SUT(X_{data}, Y_{data})) = \text{IntersecPoint}(SUT(X_{PER}, Y_{PER}))$$

HMR_{3-ROT} - Rotation of Fitted Lines . Let the SUT provide the best fitting lines, denoted as L_1 and L_2, based on the input data X_{data} and Y_{data}. The intersection point of these lines is represented as (x_{int}, y_{int}).

IF the lines L_1 and L_2 are rotated around the intersection point (x_{int}, y_{int}) between the data points X_{data} and Y_{data},

THEN the sum of squared errors (SSE) for the rotated lines ($SSE_{rotated}$) is always larger than the SSE for the original lines ($SSE_{original}$).

The core concept of HMR_{3-ROT} involves the orthogonal rotation of the output lines, which are shown in green in Figure 17. Following this rotation, an auxiliary program calculates SSE between the resulting rotated output and the input coordinates X_{data} and Y_{data}. This calculation is performed for each variation of the rotated lines. A key aspect of this HMR is that it remains no violated as long as the SEE for the rotated lines exceeds the SEE for the original lines, indicating that the original output configuration is optimally aligned with the input data. It is important to highlight that HMR_{3-ROT} differs from the conventional approach of MRs, which typically involve transforming the input data. Instead, HMR_{3-ROT} seeks to alters the SUT's output while keeping the input data unchanged, and

use the output as input to the an auxiliar function. This distinction underscores HMR_{3-ROT}'s innovative approach to verifying the SUT's reliability and accuracy of the intersection point. To elucidate this concept further, Figure 18 illustrates a conceptual implementation of HMR_{3-ROT}.

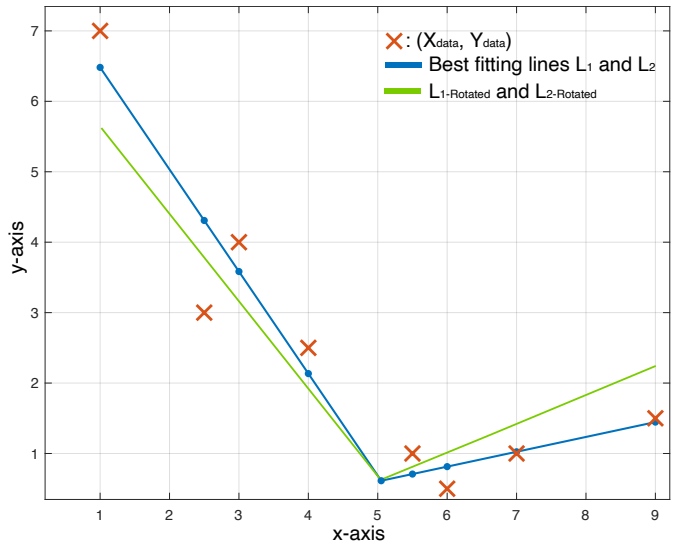

Figure 17. Graphical representation of HMR_{3-ROT}.

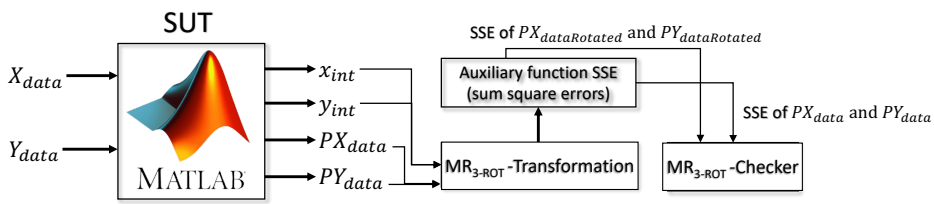

Figure 18. Conceptual implementation of HMR_{3-ROT}.

Figure 19a showcases the orthogonally rotated outputs of L_1 and L_2, rotating a range from -15 degrees to 15 degrees in $1-$degree increments. Figure 19b then plots the SEE resulting from these rotations. The $x-$axis denotes the rotation angle in degrees, and the $y-$axis quantifies the SSE. Remarkably, the graph highlights that the lowest SEE value is achieved at a $0-$degree rotation corresponding to the unaltered lines L_1 and L_2. This minimal error underscores the accuracy of the original line fittings and validates the SUT's performance through the lens of HMR_3.

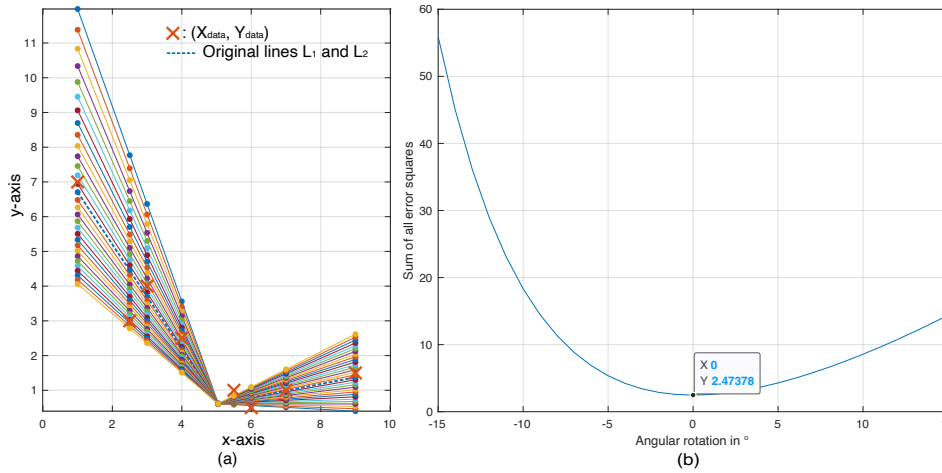

Figure 19. Analysis of Orthogonal Rotation Invariance MR_1. (a) Rotated lines of L_1 and L_2 from -15 degrees to 15 degrees in steps of 1 degree. (b) The sum of all error squares from -15 degrees to 15 degrees

7.3.2. Generic Metamorphic Relations

We refer to GMRs as a set of MRs that are derived from 'universal or generic' properties. This implies that while originally conceived in different contexts, these MRs exhibit similarities in terms of input types or internal behaviours, making them indirectly applicable to different SUTs. Our motivation for employing the term 'GMR' lies in highlighting the versatility and adaptability of these relations, which can be leveraged across diverse domains and applications.

For our SUT, we have selected six GMRs that have been applied in various contexts, such as array operations [87] and linear regression properties [14], among others. It is essential to understand that due to the generic nature of GMRs, the expected change in output resulting from a modification in input can vary. For instance, the GMR_1- Scaling aims to examine the scaling property of the SUT by multiplying the input TD, *i.e.,* X_{data} and Y_{data}, by a constant $k > 1$. In this GMR, there are four potential expected changes in the outputs, the output could be scaled by the same constant, be greater, or be less than the original, and since the constant is $k > 1$ it is known that the output cannot be equal to the original. Therefore, we consider a range of output changes that may occur. Within our selection of six GMRs, each GMR comprises sub-GMRs that explore the potential expected behaviours in more detail.

Let $X_{data} = x_1, x_2, ..., x_n$ and $Y_{data} = y_1, y_2, ..., y_n$ represent the original input data points. Let (x_{int}, y_{int}) be the intersection point of lines L_1 and L_2 calculated by the SUT. Based on this, the detailed descriptions of the GMRs are provided in the following sub-sections.

$GMR_{1.1-SCL_1}$ - *Scaling.* **IF** all input data points in X_{data} and Y_{data} are scaled by a constant $k > 1$ to obtain

$X'_{data} = k*x_1, k*x_2, ..., k*x_n$ and $Y'_{data} = k*y_1, k*y_2, ..., k*y_n$

THEN using X'_{data} and Y'_{data} as transformed inputs to the SUT, the following potential relations are produced:

- $\text{GMR}_{1.1-SCL_1}$: The resulting intersection point of lines L'_1 and L'_2 could be scaled by the same constant k, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = k* \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{1.2-SCL_>}$: The resulting intersection point of lines L'_1 and L'_2 could be greater than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) > \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{1.3-SCL_{\neq}}$: The resulting intersection point of lines L'_1 and L'_2 should be different than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data}))! = \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{1.4-SCL_<}$: The resulting intersection point of lines L'_1 and L'_2 could be lower than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) < \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

GMR_2 - Exclusive. **IF** a random pair elements (x_i, y_i) contained in the data points in X_{data} and Y_{data} is removed creating $X'_{data} = x_1, x_2 ..., x_{n-1}$ and $Y'_{data} = y_1, y_2 ..., y_{n-1}$

THEN using X'_{data} and Y_{data} as transformed inputs to the SUT, the following potential relations are produced:

- $\text{GMR}_{2.1-EXC_>}$: The resulting intersection point of lines L'_1 and L'_2 could be greater than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) > \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{2.2-EXC_<}$: The resulting intersection point of lines L'_1 and L'_2 could be lower than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) < \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{2.3-EXC_=}$: The resulting intersection point of lines L'_1 and L'_2 could be equal than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

GMR_3 - Inclusive. **IF** a random pair elements (x_{n+1}, y_{n+1}) included/added to the existing data points X_{data} and Y_{data}, creating $X'_{data} = x_1,\ x_2\ ...,\ x_n,\ x_{n+1}$ and $Y'_{data} = y_1,\ y_2\ ...,\ y_n,\ y_{n+1}$

THEN using X'_{data} and Y_{data} as transformed inputs to the SUT, the following potential relations are produced:

- $\text{GMR}_{3.1-INC_>}$: The resulting intersection point of lines L'_1 and L'_2 could be greater than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) > \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{3.2-INC_<}$: The resulting intersection point of lines L'_1 and L'_2 could be lower than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) < \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

- $\text{GMR}_{3.3-INC_{=}}$: The resulting intersection point of lines L'_1 and L'_2 could be equal than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

GMR_4 - Inverse. **IF** each element x_i and y_i contained in the data points in X_{data} and Y_{data} is inverted, resulting in $X'_{data} = \frac{1}{x_1}, \frac{1}{x_2}, ..., \frac{1}{x_n}$ and $Y'_{data} = \frac{1}{y_1}, \frac{1}{y_2}, ..., \frac{1}{y_n}$

THEN using X'_{data} and Y_{data} as transformed inputs to the SUT, the following potential relations are produced:

- $\text{GMR}_{4.1-INV_{>}}$: The resulting intersection point of lines L'_1 and L'_2 could be greater than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) > \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{4.2-INV_{<}}$: The resulting intersection point of lines L'_1 and L'_2 could be lower than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) < \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{4.3-INV_{=}}$: The resulting intersection point of lines L'_1 and L'_2 could be equal than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

GMR_5 - Reflection. **IF** the data points in X_{data} and Y_{data} are reflected, resulting in X'_{data} and Y'_{data} where:

$$X'_{data} = \{-x_i | x_i \in X_{data}\}$$

$$Y'_{data} = \{-y_i | y_i \in Y_{data}\}$$

THEN the intersection point obtained from $SUT(X'data, Y'data)$ is equal to the negative of the intersection point obtained from $SUT(X_{data}, Y_{data})$. This is:
$\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = -\text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

GMR_6 - Duplication. **IF** a random element x_i from X_{data} and its corresponding element y_i from Y_{data} are duplicated, resulting in $X'data$ and $Y'data$ where:

$$X'_{data} = x_1, x_2, ..., x_i, x_i, ..., x_n$$

$$Y'_{data} = y_1, y_2, ..., y_i, y_i, ..., y_n$$

Where x_i is duplicated element and y_i the corresponding element in $y-$coordinated.
THEN using X'_{data} and Y'_{data} as transformed inputs to the SUT, the following potential relations are expected:

- $\text{GMR}_{6.1-DUP_{>}}$ the resulting intersection point of lines L'_1 and L'_2 could be greater than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) > \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$
- $\text{GMR}_{6.2-DUP_{<}}$ the resulting intersection point of lines L'_1 and L'_2 could be lower than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) < \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

- $GMR_{6.3-DUP_{=}}$ the resulting intersection point of lines L'_1 and L'_2 could be equal than the original, this is:
  $\text{IntersecPoint}(SUT(X'_{data}, Y'_{data})) = \text{IntersecPoint}(SUT(X_{data}, Y_{data}))$

7.3.3. Experiment Procedure

Our approach to assess MR effectiveness comprises three phases. *Phase 1 - Applying MetaTrimmer* is responsible for selecting and constraining, if necessary, the HMRs and the GMRs. With this initial phase, we aim to assess the applicability of each MR from both sets, *i.e.,* HMR and GMR, to the SUT. *Phase 2 - Mutation Testing* is responsible for the generation of mutated versions of the SUT, verification of their executability, and initial analysis concerning mutation testing, *i.e.,* determining the number of killed, survived, and equivalent mutants. *Phase 3 - Assessing the Effectiveness of HMRs and GMRs* is in charge of evaluating how effectively the selected and constrained HMRs and GMRs can identify bugs when using the mutated files generated in Phase 2. Below we describe them in detail:

Phase 1 - Applying MetaTrimmer. The goal of Phase 1 is to assess the applicability of MRs within the sets of HMRs and GMRs using MetaTrimmer. Figure 20 shows MetaTrimmer contextualised within the SUT. As previously detailed in Section 5.3, MetaTrimmer is composed of three main steps: 1.1) TD Generation, 1.2 MR Process, and 1.3) MR Analysis. In the context of our SUT, 1.1) TD Generation is tasked with creating TD, denoted by X_{data} and Y_{data} in Figure 20. These represent two vector or list structures that can contain either float or integer values, with a size range from 4 to 16.

The 1.2) MT Process takes the previously generated TD and applies the corresponding transformations as indicated by each of the HMRs and GMRs. Following this, the TTD for each HMR/GMR is executed against the SUT to verify if the relations are upheld, providing a Violation Status (VS). For each HMR and GMR, a log file is created to store the TD, TTD along with the respective outputs, *i.e.*, the outputs generated by the TD and TTD and the VS. Finally, 1.3) MR Analysis investigates the frequency of violations and non-violations for each HMR/GMR.

The frequency of MR violations or non-violations is examined to determine their applicability to the SUT. A consistent 100% violation rate indicates that the MR does not apply to the SUT, while a non-violation rate suggests a match with the SUT behaviour. In scenarios with mixed cases, *i.e.,* where violations and non-violations are not 100%, it becomes crucial to identify specific TD or ranges where the MR is applicable. This analysis in guided by the following questions:

- What is the valid input space in which the HMR/GMR is violated or not violated?
- What is the input space that results neither violates nor non-violation of HMR/GMR, but causes the system to crash?

Phase 2 - Mutation Testing. The goal of Phase 2 is to generate mutated versions of the SUT and ensure that these versions are executable. Figure 21 illus-

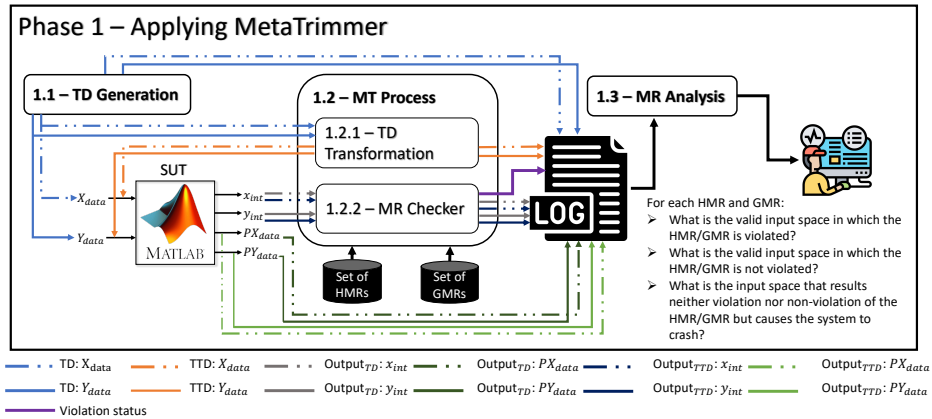

Figure 20. Overview of Phase 1 - MetaTrimmer Process in the SUT context, illustrating the three-step approach: TD Generation, MR Process, and MR Analysis.

trates the procedural flow of Phase 2, which is structured into three steps: 2.1) Automatic injection of mutants, 2.2) Verification of the mutated versions' executability, and 2.3) Initial mutation analysis. In Step 2.1) Automatic injection of mutants, the task is to create mutated versions of the SUT using MATmut, a specialised tool for mutation testing in MATLAB. Step 2.2) Verification of the mutated versions' executability, involves using the TD created in step 1.1 of Phase 1 to execute against each mutated version and recording all resulting outputs. The subsequent Step 2.3) Mutation analysis, contains two activities: 2.3.1) Filtering out versions that crash from those that do not, followed by an initial mutation analysis in the activity 2.3.1). This analysis in guided by the following questions:

- How many mutated versions have been generated?
- What is the ratio of crashes to non-crashes across all the TD?
- How many mutants are killed or survived?

Phase 3 - Assessing the Effectiveness of HMRs and GMRs. Phase 3 centres on determining the effectiveness of the HMRs and GMRs in terms of bug-finding capabilities using the mutated versions generated in Phase 2. As it shows Figure 22, it comprises three steps: 3.1) Execution of TD and TTD for each HMR/GMR against non-crashing mutated versions, 3.2) Acquisition of the violation status for each HMR/GMR, and 3.3) Analysis of Violation/Non-Violation instances.

The step 3.1) involves executing both TD and TTD, which were prepared in Phase 1, against each non-crashing mutated version of the SUT. This step is critical in determining the response of each mutated version when tested with the HMRs and GMRs. The execution process is thorough, and each resulting output is recorded. In step 3.2, the violation status for each HMR/GMR is obtained, and the focus shifts to determining whether the HMRs and GMRs are violated or not. The MR Checker of Phase 1 is used to systematically assess and store the sta-

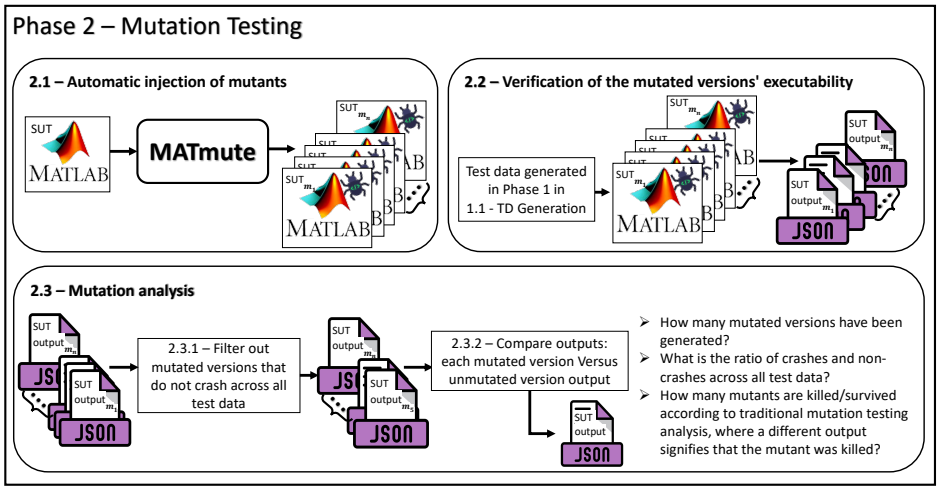

Figure 21. Overview of Phase 2 - Mutation Testing, depicting the procedural flow across three main steps: Automatic Injection of Mutants, Verification of Mutated Versions' Executability, and Analysis of Killed and Survived Mutants.

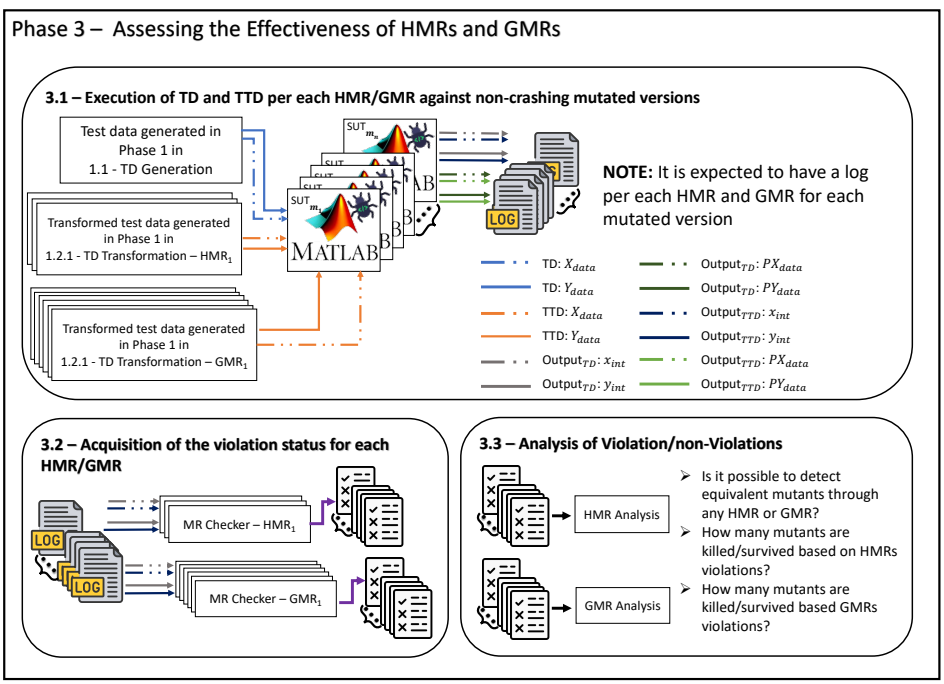

Figure 22. Overview of Phase 3 — Assessing the effectiveness of HMRs and GMRs, showing the three-step process - Execution of TD and TTD per each HMR/GMR against non-crashing mutated versions, acquisition of violation status per each HMR/GMR, and analysis of violation and non-violation instances.

tus of each relation. The step 3.3) Analysis of Violation/Non-Violation Instances entails a detailed review of the violations and non-violations stored in the previous step. This analysis aims to ascertain the MRs' bug detection capabilities. In evaluating the effectiveness of HMRs and GMRs, we adopt a three-tiered assessment approach inspired by principles of mutation testing. Our approach begins by identifying mutants that have survived all TD inputs. We believe that upon detailed examination, these mutants have a high probability of being equivalent mutants, which do not alter the program's output despite changes in the code. This first level of our assessment focuses on determining the ability of the HMRs and GMRs to detect undesirable behaviours in these groups of equivalent mutants. Specifically, we evaluate whether any HMRs/GMRs can indicate potential issues by identifying violations of the expected behaviours in these mutated SUT.

The second level of our assessment approach represents the opposite scenario. Here, we identify those mutated versions for which the mutant has been 'killed' across all TD inputs. This means that every TD input has successfully detected errors or abnormal behaviour in these mutants, indicating that the alterations in the code significantly impact the program's functionality. Specifically, in this level, we are interested in determining whether any of the HMRs/GMRs can perform at least as well as mutation testing in terms of bug finding. We aim to evaluate the effectiveness of each rule by analysing the violation rate for each HMR and GMR. This metric provides crucial insights into how frequently each relation flags a mutation as a deviation from expected behaviour, thereby reflecting its sensitivity and reliability in detecting errors.

The third level of our assessment examines 'mixed cases', where mutants exhibit varying responses depending on the TD inputs. In these scenarios, a mutant might be 'killed' by some TD inputs while surviving others. Here, we focus on quantifying the amount of TD required to kill the mutants instead of the amount needed to merely trigger a rule violation within each HMR and GMR. By investigating these mixed cases, we seek to understand the relative sensitivity and robustness of each HMR/GMR.

7.3.4. MATmute - Mutation Testing Tool for MATLAB

MATmute[1] is an open-source mutant generator developed by Hook *et al.* [57] that is designed for MATLAB code. When provided with a target MATLAB function or file, MATmute systematically applies mutation operations to generate a collection of mutants. These operations include:

- *Statement deletion*: A statement is commented out.
- *Branch negation*:A branch condition is negated, forcing the opposite decision.
- *Constant replacement*: Hard-coded constants are replaced with others.

[1] MATmute

- *Operator replacement*: Mathematical operators are substituted with different ones.
- *Assignment perturbation*: The right-hand side of an assignment statement is multiplied by a constant before completing the assignment.

MATmute comprises two primary components, a Python module responsible for generating mutants, *i.e.,* `.m` files, and a MATLAB module for executing these mutants and computing mutation sensitivity scores. Subsequently, the tool generates a comparison graph illustrating mutation scores for corresponding test cases.

To use MATmute, Python 2.5 or 2.6 and MATLAB must be installed on the system. The Python module, named pymute, accepts a MATLAB script as input and proceeds to generate mutants for the function. These mutants are stored in a designated folder created upon pymute execution. Pymute consists of six submodules, which are detailed below:

1. `MATmute`: It is the main file and it is responsible for orchestrating the mutation process.
2. `Mutatee`: Cleans the code file intended for mutation.
3. `Operators`: Stores the mutation operators to be applied to the code.
4. `Mutants`: Generates mutant files with the assistance of the Mutator.
5. `Ops_config`: Defines the mutation operators within this file.
6. `Mutator`: Receives the code and mutation operators, generating mutations accordingly.

The overall working process is as follows; Initially, the `MATmute` sub-module calls the `Mutatee` sub-module to clean and store each statement of the target code. Then, the code is instrumented and stored. `MATmute` calls `Operators` to obtain the set of mutation operators to be applied to the target code. `MATmute` provides `Operators` and the `Mutatee` object to `Mutants` and requests all possible mutations of the code. `Mutants` utilizes the `Mutator` to generate mutants. The `Mutator` invokes `Operators` to generate mutants according to the configurations in `Ops_config`. Finally, `MATmute` instructs `Mutants` to generate mutant files and store them in a separate directory.

7.4. Results

This section details the implementation details and the key findings from each phase. As mentioned in Section 7.2.2, due to confidentiality agreements with our industry partner, we are not permitted to share the source code of the SUT or the mutated versions generated. However, we provide template scripts that can be utilised to replicate our method with any MATLAB program. Moreover, we have made the complete data generated during our experiments available in our GitHub repository[2].

7.4.1. Phase 1 - Applying MetaTrimmer

Figure 23 illustrates the pipeline that was developed for implementing MetaTrimmer and conducting the evaluations. The implementation consists of five Python scripts and one MATLAB script. The first script, `InputGenerator.py`(1), designed to automate the creation of the TD. This script generates two sets of data, X_{data} and Y_{data}. Each set consists of a random number of integers and floats. The size of each set, as well as the range of numbers, is determined by a random selection within predefined limits—specifically, $[min, max]$, which are set to 4 and 16 for the size, and $[0, 10]$ for the number range. The X_{data} and Y_{data} vectors are shuffled to ensure a random distribution of the elements. The output of this script is a `JSON`(a) file that contains the generated TD. We have ensured a uniform distribution with respect to the sizes of each vector; each vector size is represented with 15 instances, hereinafter TD inputs, resulting in a total of 195 TD inputs. For example, for $X_{\text{data}_1} = [X_1, X_2, X_3, X_4]$ and its corresponding $Y_{\text{data}_1} = [Y_1, Y_2, Y_3, Y_4]$, the size of the vector is 4, and there are 14 more TD inputs of vectors of this size. This uniformity is maintained for each vector size ranging from 4 to 16, amounting to 195 TD inputs in total.

Table 16 summarised basic statistics for X_{data} and Y_{data}, each set comprises 195 individual vectors, set of TD inputs, whose lengths vary from a minimum of 4 to a maximum of 16 elements. The 'Min sum', 'Avg sum', and 'Max sum' columns reflect the aggregated totals of the elements within each vector across the TD. For X_{data}, the vectors have a minimum sum of elements at 10.8, an average sum at 49.82, and a maximum sum reaching 100.7. Meanwhile, Y_{data}'s vectors show a slightly lower minimum sum of 6.9, an average sum very close to X_{data} at 49.41, and a maximum sum at 99.2. We maintain a uniform distribution with respect to the vector's size. Each possible size from 4 through 16 is uniformly represented by 15 samples. For instance, size 4 is associated with 15 distinct vectors, a pattern that consistently applies to subsequent sizes up through size 16.

The second script, `InputTransformer.py`(2.1), transforms the TD generated by `InputGenerator.py` based on the rules defined by each HMR and GMR. The input transformer script takes the `JSON`(a) file containing the generated TD

[2] Assessing-the-Strength-of-Metamorphic-Testing Repo

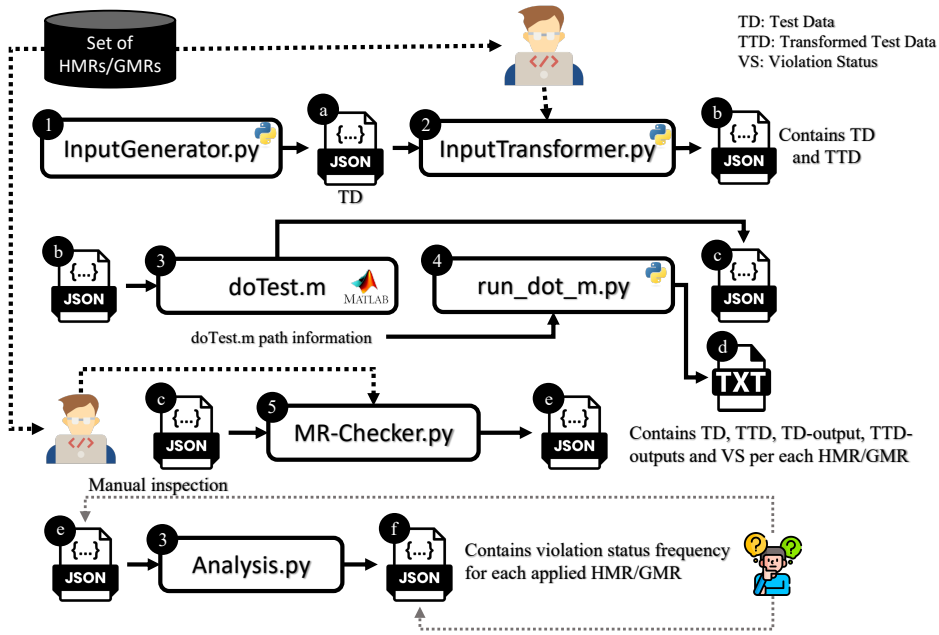

Figure 23. Implementation pipeline of MetaTrimmer for phase 1

	Instance count	Min size	Max size	Min sum	Avg sum	Max sum
X_{data}	195	4	16	10.8	49.82	100.7
Y_{data}	195	4	16	6.9	49.41	99.2

Table 16. Summary statistics of the test data inputs generated, *i.e.,* X_{data} and Y_{data}

as input and produces a new `JSON`ⓑ file with the TTD as output. It is important to note that the MRs are described in natural language and need to be translated into code so that they can be applied to the TD. The HMRs and GMRs are hard-coded in `InputTransformer.py` script so that they are applied automatically to the generated TD to generate the TTD. We have created templates so we can use them at anytime, and for different SUTs.

The third script, `doTest.m`③ is tasked with configuring the parameters necessary for the SUT. In addition, the script reads input TD and TTD from a JSON file, processes each TD according to its key, and applies it to the SUT for execution. Specifically, the script handles data extraction, execution of the SUT with both TD and TTD, and the storage of the generated outputs into a structured format, encoding this data as `JSON`ⓒ. To complement the `doTest.m` script, which is a MATLAB script, the script `run_dot_m.py`④ serves as a bridge to automate the invocation of `doTest.m` within a MATLAB environment from a Python context. The `run_dot_m.py` script employs the `subprocess` module to execute MATLAB commands, managing the execution of the `doTest.m` script for various sets of

SUT version. It begins by capturing the start time, followed by the execution of the MATLAB script with the necessary command options to ensure that MATLAB runs in a non-interactive mode. This approach ensures minimal overhead and a focus on script execution. The standard output and errors from the MATLAB process are redirected to an `txt`(d) file, enabling the capture and review of the script's execution log.

The `MR-Checker.py`(5) script inputs are the outputs of `doTest.m` script, *i.e.,* `JSON`(c), and checks whether the outputs of the TD and the TTD match the corresponding HMR/GMR relation. Similar to `InputTransformer.py` script, the expected output relations for each HMR/GMR are also hard-coded into the `MR-Checker.py` script, which means that the script applies these relations automatically to the generated outputs from the executer scripts. After the TD is generated, transformed, and checked, a `JSON`(e) file is produced per each HMR/GMR containing information about the TD, the TTD, their respective outputs, and the verdict of the MR-Checker for each HMR/GMR regarding the violation status. The `JSON`(e) file produced by the MR-Checker are used as input for the final script, called `Analiser.py`(6). The `Analiser.py` script calculates for each method how often each of the applied MRs is violated or not violated, and store this information, `JSON`(f) in Figure 23.

Table 17 presents the count and percentage of TD inputs that resulted in non-violations, violations, and crashes. Figure 24 provides a visual representation of this data, depicted through coloured bars: non-violations are illustrated with blue bars, violations with orange, and system crashes with green. For the HMR set, the results indicate that HMR_{1-SHT} and HMR_{-PER} had a 100% rate of non-violation with all 195 inputs, showing no TD produced a violation or system crashes. In contrast, $HMR3-ROT$ displays a mixed outcome with 59% non-violations and 41% violations. This suggests that in those 80 cases, the *SSE* for the rotated lines was not larger than the *SSE* for the original lines, contrary to the expectation set by $HMR3-ROT$. Therefore, we examined in detail the 80 TD inputs that resulted in violations in HMR_{3-ROT}.

Our examination began by analysing the rotation degrees for each input TD where rule violations occurred. Figure 25 shows the distribution of the rotation degrees for each TD inputs where the HMR_{3-ROT} violated. As Figure 25 there does not seem to be a clear pattern or correlation between the degree of rotation and the occurrence of rule violations. This indicates that the extent of the rule violation (if at all) is not dependent on the angle by which the lines are rotated. This suggests that other factors may be contributing to the instances where HMR_{3-ROT} is violated.

Figure 26 and Figure 27 visualise the absolute and relative differences in SSE along with their frequency, with each bar representing a violation instance and the scatter points depicting the associated rotation angles. Upon closer inspection, and in consultation with subject matter experts, it is observed that the improvements, or rather the decreases in SSE due to rotation (which should not happen accord-

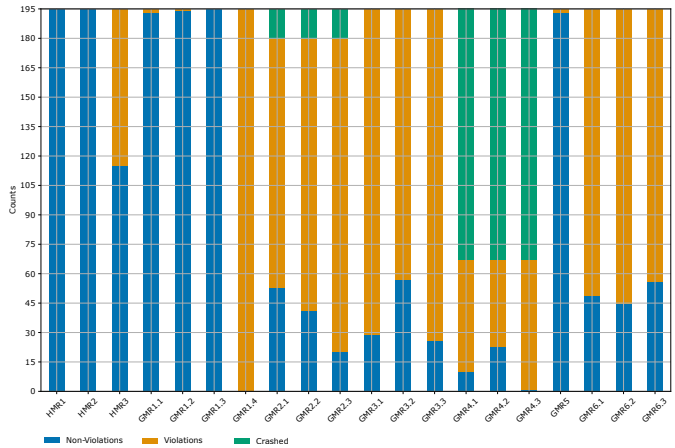

Figure 24. Frequency of non-violations, violations, and system crashes per HMR and GMR

Table 17. Count of non-violations, violations, and crashes observed for each HMR and GMR, alongside calculated percentages for each outcome category

	GMRs					
	Non-Violations		**Violations**		**Crashed**	
	Total # of inputs	[%]	Total # of inputs	[%]	Total # of inputs	[%]
$GMR_{1.1-SCL_1}$	194	99	1	1	0	0
$GMR_{1.2-SCL_>}$	194	99	1	1	0	0
$GMR_{1.3-SCL_{\neq}}$	195	100	0	0	0	0
$GMR_{1.4-SCL_<}$	0	0	195	100	0	0
$GMR_{2.1-EXC_>}$	53	27	127	65	15	8
$GMR_{2.2-EXC_<}$	41	21	139	71	15	8
$GMR_{2.3-EXC_=}$	20	10	160	82	15	8
$GMR_{3.1-INC_>}$	29	15	166	85	0	0
$GMR_{3.2-INC_<}$	57	29	138	71	0	0
$GMR_{3.3-INC_=}$	26	13	169	87	0	0
$GMR_{4.1-INV_>}$	10	5	57	29	128	66
$GMR_{4.2-INV_<}$	23	12	44	23	128	66
$GMR_{4.3-INV_=}$	1	1	66	34	128	66
GMR_{5-REF}	195	100	0	0	0	0
$GMR_{6.1-DUP_>}$	49	25	146	75	0	0
$GMR_{6.2-DUP_<}$	45	23	150	77	0	0
$GMR_{6.3-DUP_=}$	56	29	139	71	0	0

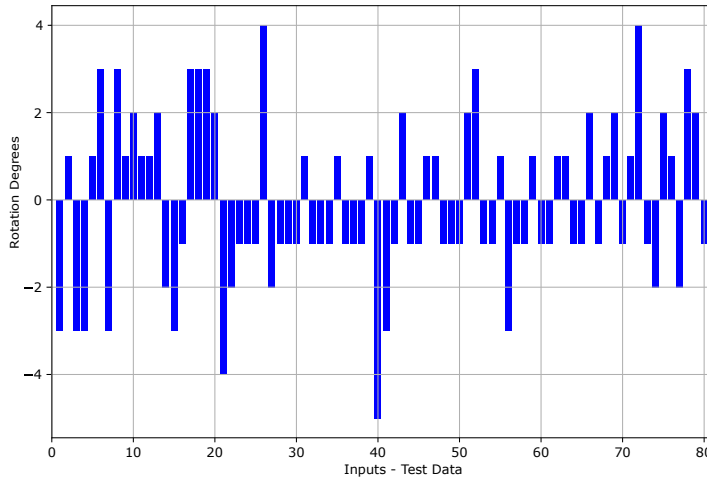

Figure 25. Distribution of rotation degrees for each TD input set where HMR_{3-ROT} is violated. No discernible correlation is observed between the rotation angles and the rule violations, indicating the non-dependency of SSE changes on rotation degrees

ing to HMR_{3-ROT}), are not statistically significant. Most relative differences in SSE, as indicated by the frequency distribution in the Figure 27, fall below a 1% improvement. This marginal improvement could be attributed to rounding errors, the inherent noise within the data, or other imprecision in the SUT.

The approach to addressing violations of HMR_{3-ROT} in subsequent analyses is to treat them as distinct from the remaining input TD, recognising that while the rule is not upheld across all TD, the SUT may still function reliably. This leads us to propose a concept of 'softened' MRs, which acknowledges small deviations from the expected rule as acceptable in practice. If the SUT is unaffected by these small violations, it supports the idea that HMR_{3-ROT} is practically, though not strictly theoretically. This 'softened' MRs could suggest introducing a tolerance threshold, for instance, in the case of HMR_{3-ROT}, introducing a tolerance threshold for SSE increases after line rotation, within which the SUT's output is still considered accurate.

In regard to the GMR set, the only GMRs that exhibit high rates $(>= 99\%)$ of non-violations are $GMR_{1.1-SCL}$, $GMR_{1.2-SCL>}$, $GMR_{1.3-SCL\neq}$, and GMR_{5-REF}, whereas $GMR_{1.4-SCL<}$ shows a 100% violation rate. GMR_{2-EXC} and $GMR_3 - INC$ and sub-categories present a diverse range of results, with substantial numbers of violations and non-violations, while also including some instances where the system crashed. Particularly, in cases where crashes occurred, such as with $GMR_{2.1-EXC_>}$, $GMR_{2.2-EXC_=}$, and $GMR_{2.3-EXC_<}$, those crashes account for 8% of the outcomes. Upon inspecting the reasons behind these crashes, it turned out that the change statement of $GMR_2 - EXC$ generates invalid data. All 15 input TD that crashed have a size of 4; therefore, after the transformation, they will have a size of 3. In this scenario, such invalid data resulting from crashes is considered correct outcomes.

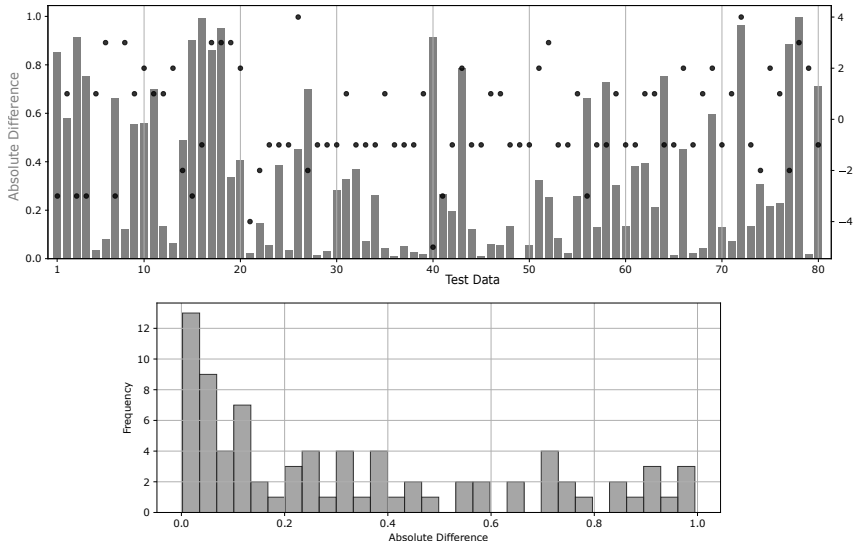

Figure 26. Bar and scatter plot of the absolute differences in SSE juxtaposed with rotation degrees. The data indicates that most rule violations have an inconsequential impact on the SSE values, aligning with expert feedback that these deviations lack substantive significance

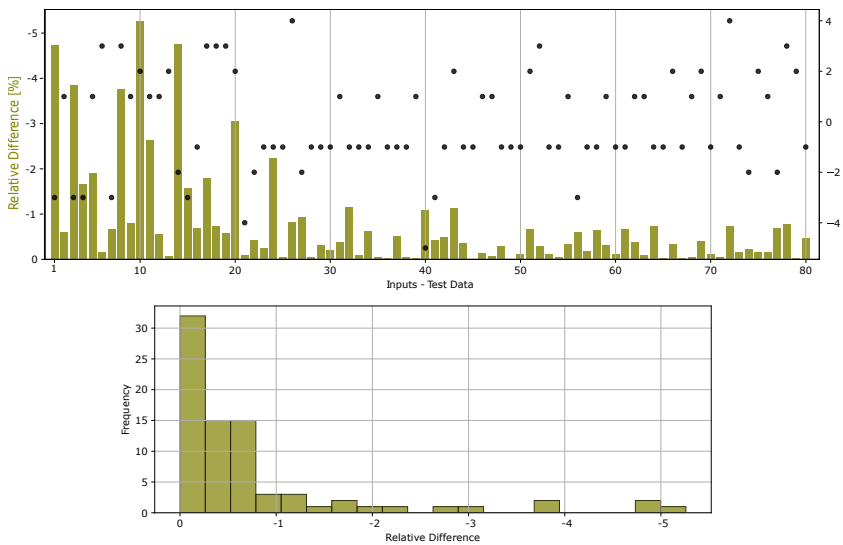

Figure 27. Bar and scatter plot illustrating the relative differences in SSE along with their frequency. The histogram emphasises that the majority of SSE improvements due to rotation are under 1%, reinforcing the conclusion that HMR_{3-ROT} violations are typically of minimal practical consequence

GMR_{4-INV} has a substantial number of inputs that lead to crashes, especially $\text{GMR}_{4.1-INV_{>}}$ to $\text{GMR}_{4.3-INV_{=}}$, where crashes occur for two-thirds of the inputs. Upon inspection, we found that all 128 TD inputs contain elements equal to zero. This is crucial because the transformation specified by GMR_{4-INV} in-

volves taking the inverse of each element. Consequently, when attempting to compute the inverse of elements that are zero, the transformation yields division by zero errors, resulting in crashes. GMR_{5-REF} shows a stable outcome with 99% non-violations, while $GMR_{6.1-DUP_{>}}$, $GMR_{6.2-DUP_{<}}$, and $GMR_{6.3-DUP_{=}}$ indicates higher violation rates. Upon examining non-violations versus violations, no discernible pattern is found to explain the reasons behind the violations in the GMRs of GMR_{2-EXC}, GMR_{3-INC}, GMR_{4-INV}, and GMR_{6-DUP}. Given that these GMRs exhibit substantially more violations or crashes, one can assume that these GMRs do not apply to the SUT. In contrast, regarding the GMRs with high rates ($>= 99\%$) of non-violations, such as $GMR_{1.1-SCL_1}$, $GMR_{1.2-SCL_{>}}$, $GMR_{1.3-SCL_{\neq}}$, and GMR_{5-REF}, it is assumed that they do apply to the SUT. The 1% of violations observed for $GMR_{1.1-SCL_1}$ and $GMR_{1.2-SCL_{>}}$ were also examined. These violations stem from the scaling process, which generates a new intersection point in negative Cartesian coordinates. It is evident that such cases will be violated due to the transformation applied. Consequently, this particular TD input is expected to consistently result in violations.

7.4.2. Phase 2 - Mutation Testing

As discussed in Section 7.3.3, the objective of phase 2 is to generate mutated versions of the SUT and ensure their executability. For the automatic injection of mutants, we employ MATmute, an open-source mutant generator designed for MATLAB. Notably, this tool has not undergone updates since 2009 and relies on Python 2.5. To ensure its proper functioning, we utilise a Docker container to install Python 2.5 and exclusively employ MATmute for generating mutated files without any execution. This approach is necessitated by the lack of MATLAB support within the Docker container, as MATLAB execution would require MATLAB itself. For generating the mutated SUT version we used the default mutants types that the MATmute provides. We have made the Docker container containing MATmute available in our GitHub repository[3]. This container includes all the necessary dependencies and instructions for performing mutation testing on MATLAB code using MATmute.

Table 18 provides a comprehensive summary of the mutation testing phase results, showcasing the outcomes for different types of mutations applied to the programs. Each row corresponds to a specific type of mutation, including constant replacement, operator replacement, statement deletion, and negated branch expression. For each mutation type, the table indicates the number of mutated programs generated, the count of programs that resulted in crashes, and the instances where any TD input did not lead to a crash.

[3]MATmute-Docker Repo

Table 18. Summary of Mutated Programs Generated

Type of Mutation	Mutated Programs Generated	Crashed for all TD	Non-Crash for Any of the TD
Constant Replaced	780	433	192
Operator Replaced	514	126	164
Statement Deleted	91	31	49
Negated Branch Expression	5	2	0
Total of Mutated Programs	**1390**	**592**	**405**

Table 19. Summary of Assessment Levels for HMRs and GMRs Effectiveness and the Total of Mutated SUT Versions per Level in

Level	Description	Focus of Analysis	MSV*
1	Analysis of mutants that have 'survived' all TD inputs (equivalent mutants)	The goal is to determine the ability of the HMRs and GMRs to detect undesirable behaviours in these groups of equivalent mutants.	236
2	Analysis of mutants that have been 'killed' by all TD inputs.	The goal is to assess if the HMRs/GMRs can perform at least as well as traditional mutation testing in terms of bug finding.	85
3	Analysis of 'mixed cases', where mutants show variable responses to different TD inputs — being killed by some while surviving others.	The goal is to quantify the amount of TD necessary to kill the mutants versus the amount needed to trigger a rule violation in each HMR and GMR.	84

MSV*: Mutated SUT Versions

7.4.3. Phase 3 - Assessing the Effectiveness of HMRs and GMRs

From Phase 2 - Mutation Testing, we obtained 405 mutated versions of the SUT, all of which remained crash-free when subjected to TD inputs. Phase 3 focuses on assessing the effectiveness of the HMRs and GMRs in terms of their bug-finding capabilities using these mutated SUT versions. To evaluate this, we followed the procedure outlined in Section 7.3.3, as depicted in Figure 22. Leveraging the TD and TTD generated in Phase 1 for each HMR and GMR, we executed them against the 405 mutated SUT versions, resulting in individual log files for each rule. Subsequently, we analysed these log files to ascertain the violation status based on the expected relations defined by each HMR and GMR. The total number of logs generated during the execution of TD and TTD for each HMR and GMR against the 405 mutated versions amounted to 2835. Considering a total of 7 MRs chosen, HMR_{1-SHT}, HMR_{2-PER}, HMR_{3-ROT}, for the HMRs and $GMR_{1.1-SCL_1}$,

Table 20. Count of Non-violations and Violations Observed for the Selected HMR and GMR in Level One, 263 Mutated SUT Versions

	HMRs					
	Non-Violations		**Violations**		**Crashed**	
	Total # of inputs	[%]	Total # of inputs	[%]	Total # of inputs	[%]
HMR_{1-SHT}	195	100	0	0	0	0
HMR_{2-PER}	195	100	0	0	0	0
HMR_{3-ROT}	115	59	80	41	0	0
	GMR					
$GMR_{1.1-SCL_1}$	194	99	1	1	0	0
$GMR_{1.2-SCL>}$	194	99	1	1	0	0
$GMR_{1.3-SCL\neq}$	195	100	0	0	0	0
GMR_{5-REF}	195	100	0	0	0	0

$GMR_{1.2-SCL_>}$, $GMR_{1.3-SCL_{\neq}}$, and GMR_{4-REF} for the group of GMRs.

As mentioned in Section 7.3.3, we assess the effectiveness of HMRs and GMRs at three distinct levels. Table 19 summarises and describes these levels, along with the total number of mutated SUT versions per level. Table 20 presents the outcome at Level 1 of our assessment, which focuses on the analysis of mutants that have survived all TD inputs. It details the count of non-violations and violations observed for the selected HMR and GMR. An important finding is that the violations and non-violations are consistent for all 236 mutated versions. For HMR_{1-SHT} and HMR_{2-PER}, all tested inputs—100%—are non-violations, which suggests that there were no deviations from the expected behaviour across all TD. For $GMR_{1.1-SCL_1}$, $GMR_{1.2-SCL>}$, and GMR_{5-REF}, we observe a similar trend to HMR_{1-SHT} and HMR_{2-PER}. In general, these results are similar to the ones presented in Table 17 from Phase 1, suggesting that both HMRs and GMRs are unable to detect any equivalent mutants. However, when inspecting HMR_{3-ROT} we found a shift in the range of improvement. Figure 28 provides a bar and scatter plot illustrating the relative differences in SSE, accompanied by their corresponding frequency. Notably, the histogram underscores that the majority of the SSE improvements due to rotation are now under 5%, a significant increase from the previous data inspection without the mutants, where improvements were around 1%. This discrepancy suggests that the introduction of mutants into the system affects the SSE improvement margin.

At Level 2, we aim to determine if the HMRs and GMRs can match the bug detection efficacy of traditional mutation testing. This level saw the evaluation of 85 mutated SUT versions, each of which was conclusively 'killed' by all TD when subjected to traditional mutation testing. The focus is primarily on the capability of HMRs and GMRs to identify and flag mutations. According to the data presented in Table 21, a 'violation' denotes an instance where at least one TD input

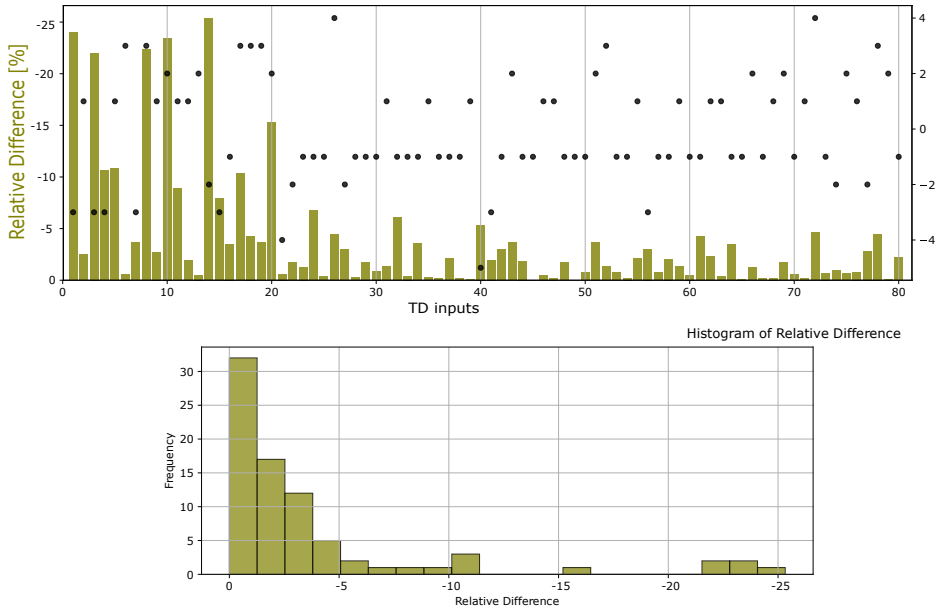

Figure 28. Bar and scatter plot illustrating the relative differences in SSE along with their frequency. The histogram emphasises that the majority of SSE improvements due to rotation are under 5%

prompted a departure from the predicted behaviour under the specific HMR/GMR. On the other hand, 'non-violations' reflect cases in which the HMR/GMR frameworks failed to detect any departure.

Table 21 lists the numbers of mutated SUT versions that triggered violations in response to at least one TD input and the count of those where no violations were observed. Unlike the previous level, in this level, it is possible to rank MRs based on their effectiveness in detecting mutants; we can use the number of violations triggered as an indicator. A higher number of violations would suggest a higher sensitivity of the MR to detect mutants, thus a more effective MR. Based on Table 21, where violations are indicated by "MV_VT" (Mutated Version With Violations Triggered), the ranking from most to least effective MR would be as follows:

1. HMR_{3-ROT} - 85 violations: This HMR is the most effective in detecting mutants, as it has the highest number of violations, implying that it has triggered for each mutated version tested.
2. $GMR_{1.2-SCL_{>}}$ and GMR_{5-REF} (tie) - 80 violations: Both GMRs are very effective, with a high number of violations close to that of HMR_{3-ROT}
3. HMR_{1-SHT} - 63 violations: This MR ranks as moderately effective, with a significant number of violations, but not as high as the top performers.
4. HMR_{2-PER} - 39 violations: With fewer violations than HMR_{1-SHT}, this MR is less effective in detecting mutants according to the data provided.

5. $GMR_{1.1-SCL_1}$ - 44 violations: Slightly more effective than HMR_{-PER}, but with fewer violations than HMR_{1-SHT}, placing it in the middle range.
6. $GMR_{1.3-SCL\neq}$ - 0 violations: This MR did not trigger any violations, suggesting that it was ineffective in detecting mutants within the tested SUT versions.

Table 21. Number of Mutated SUT Versions with Violations Triggered by at Least One TD Input, Along with Non-Violating Mutated SUT Versions

Category	HMR_{1-SHT}	HMR_{2-PER}	HMR_{3-ROT}	$GMR_{1.1-SCL_1}$	$GMR_{1.2-SCL>}$	$GMR_{1.3-SCL\neq}$	GMR_{5-REF}
MV_VT	63	39	85	44	80	0	85
MV_NVT	22	46	0	41	5	85	0

MV_VT: Mutated Version With Violations Triggered
MV_NVT: Mutated Version Without Violations Triggered

At Level 3, we explore the 'mixed cases'. These scenarios are particularly insightful as they showcase the diverse reactions of mutants to various TD inputs—some resulting in the mutant being killed, while others demonstrate survival. Our goal in this level is to quantify the volume of TD necessary to effectively kill the mutants compared to the data required to trigger a rule violation.

Figure 29 and Figure 30 offer a comparative analysis of mutant responses within 'mixed cases,' where mutants aren't consistently detected across different TD inputs. In Figure 29, the first panel illustrates the amount of data required to kill the mutant for each mutated version. The bottom panels depict the response of the three HMRs, showcasing the amount of data needed to trigger a violation and detect the mutant for each mutated SUT version. Figure 30 shows the same first panel as Figure 29, and the response of the four GMRs.

Upon analysing the behaviour patters in Figure 29 and Figure 30, we establish a hierarchical ranking of the HMR and GMR based on their ability to detect mutations and the volume of TD inputs required for detection. This enumeration aligns with the criteria outlined in Level Two, where the HMRs and GMRs are deemed effective if it triggers a violation with at least one TD input. Here, however, we emphasise HMRs and GMRs that trigger with the highest possible number of TD inputs, indicating a broader detection capability. The ranking is as follows:

1. HMR_{3-ROT} - Demonstrates the highest detection capability, triggering violations with nearly all TD inputs, positioning it as the top-performing MR.
2. GMR_{5-REF} - Demostrates extensive range of violations across TD inputs, indicating a strong ability to detect a wide array of mutations.
3. HMR_{1-SHT} - Although triggering violations with a high number of TD inputs, this MR ranks below the top performers due to slightly less detection breadth, with only 39 out of 84 mutants detected.
4. $GMR_{1.1-SCL_1}$ and $GMR_{1.3-SCL>}$ - Displays moderate detection abilities, effective with a significant number of TD inputs but not as comprehensive as higher-ranked MRs.

5. HMR_{2-PER} and $GMR_{1.3-SCL\neq}$- This HMR shows reliable detection capabilities in terms of the volume of the TD but only in very specific mutants.

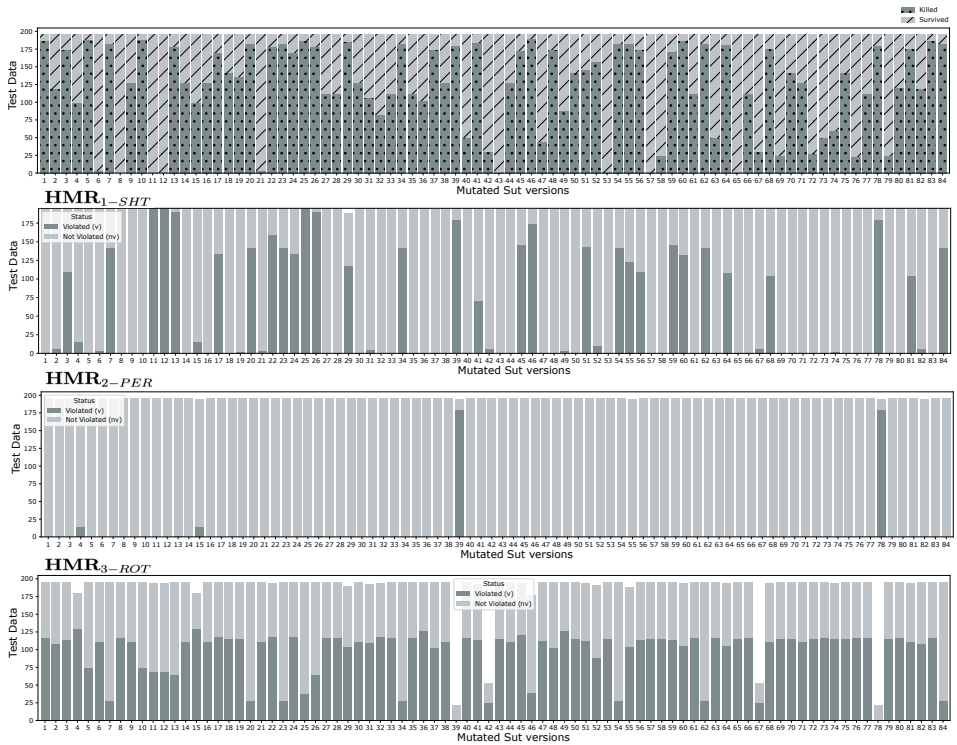

Figure 29. Comparative Analysis of Mutated SUT Versions in Mixed Cases for HMRs. The top panel illustrates the data volume required to 'kill' the mutant for each mutated version, represented by dark grey colour and black dots, while the bottom panels detail the data volume required to trigger a violation, represented by dark grey, of the three HMRs

7.5. Discussion

In this section, we delve into the key findings derived from our results, outlined in Section 7.5.1. We conclude this section by addressing the threats to the validity of our study in Section 7.5.2.

7.5.1. Key Findings

In this study answer RQ_3 by proposing a three-level strategy to assess the effectiveness of MRs. The first level investigates the MRs' ability to identify equivalent mutants—those that do not alter the program's behaviour in any observable way. The second level inversely examines MRs against mutants that have been 'killed' across all TD inputs, offering a lens into the MRs' capability to detect bugs as effectively as traditional mutation testing. The third level, delves into 'mixed cases'.

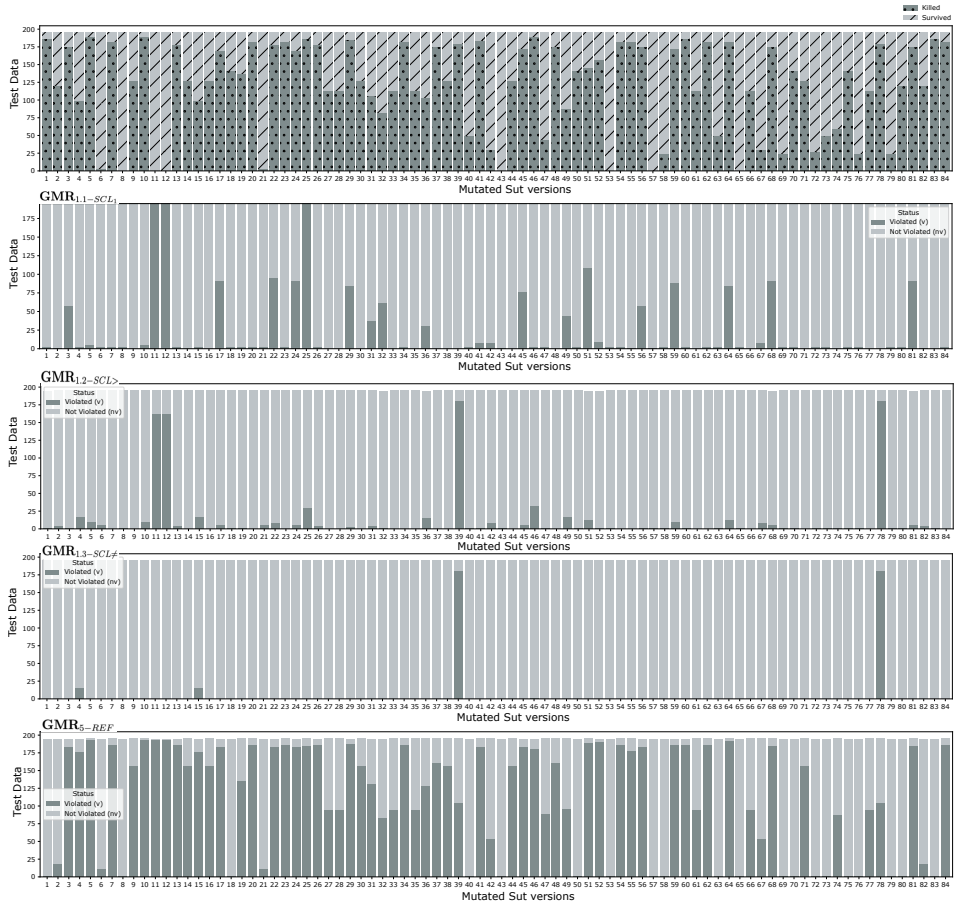

Figure 30. Comparative Analysis of Mutated SUT Versions in Mixed Cases for HMRs. The top panel illustrates the data volume required to 'kill' the mutant for each mutated version, represented by dark grey colour and black dots, while the bottom panels detail the data volume required to trigger a violation, represented by dark grey, of the four GMRs

Here, mutants exhibit variable responses to different TD inputs—some leading to mutant detection and others not. This level assesses the quantity of TD required to trigger violations from the mutated version of the SUT, providing a measure of MRs' sensitivity and breadth of detection. For our SUT, the MR that consistently outperformed others across all three levels of assessment was HMR_{3-ROT}. This rule maintained the top rank in each level, showcasing its robustness and effectiveness. From the initial phase onwards, HMR_{3-ROT} demonstrated its capability to identify instances where the "optimal solution" may not always be optimal within the context of our SUT. Additionally, GMR_{5-REF} consistently exhibited strong performance, consistently securing the second rank in the second and third levels of assessment. HMR_{2-SHT} also displayed respectable performance, consistently ranking in the third position across both the second and third levels of assessment.

Overall, our study yields three significant insights into assessing the effectiveness of MRs through mutation testing:

1. It is crucial to recognise that MRs may not detect significant changes when dealing with equivalent mutants. While this may seem obvious, the detection of equivalent mutants remains a common challenge and a hot topic within the mutation testing community.
2. Effective MRs demonstrate their efficacy when mutants are successfully killed by all TD. In such cases, an effective MR will also detect or 'kill' the mutant by triggering a violation in at least one TD.
3. An effective method for assessing MRs' effectiveness involves analysing the amount of data required to trigger a violation. Our analysis of mixed cases revealed instances where mutations were not successfully detected by certain TD, yet MRs triggered violations. This emphasises the need to evaluate MRs not only based on their ability to detect mutants but also on their sensitivity to variations across different TD.

Another important finding pertains to the concept of 'softness' within the context of MRs. The concept of 'softness' in MRs, as observed with HMR_{3-ROT}, indicates the need to introduce a certain degree of tolerance within the MRs definition. It acknowledges that not all violations may signify a fault in the SUT, and as such, softness allows permissible variations, adding flexibility to the MT process. It is important to highlight that the implementation of 'softness' requires a thorough understanding of the domain and the acceptable limits within which the system can operate without being deemed faulty.

7.5.2. Threats to Validity

In our study, three types of threats to validity are most relevant: threats to construct, internal and external validity. Construct validity relates to the appropriateness of the constructs we defined to assess MR strength and rank MRs based on fault-detection effectiveness. This threat was mitigated by adapting well-known principles from mutation testing, traditionally used to evaluate test suite adequacy. For example, mutation testing typically discards equivalent and trivial mutants because they do not contribute to identifying faults in the SUT. In our approach, however, these types of mutants (level 1 and 2) are used to identify particularly strong or weak MRs. This adaptation allows us to leverage the nuanced characteristics of MRs that mutation testing reveals, aligning with the thesis's goal of refining and ranking MRs for improved scalability and automation in MT. The remaining mutants (level 3) are then used to rank MRs in terms of their data sensitivity, providing a comprehensive evaluation framework grounded in established mutation testing practices.

A potential threat to internal validity is linked to the adequacy of the TD. The effectiveness of MRs and mutation testing is highly dependent on the input data used. If the data is not sufficiently diverse or extensive, it may not trigger all the

relevant behaviours in the SUT, leading to an underestimation of the MRs' capabilities as well as the mutation testing capabilities. Furthermore, mutation testing and MT differ fundamentally in their approaches. Mutation testing assumes that the unmutated version of the SUT is correct and relies on comparing its output to that of the mutated versions to identify incorrect behaviours. This reliance can limit the usefulness of trivial mutants—those that are semantically unrealistic—in traditional fault detection contexts. However, in MT, this limitation is inherently mitigated. Since MT does not depend on a baseline "correct" output, it instead evaluates the relation between inputs and outputs as defined by MRs. This ensures that even trivial mutants can provide meaningful insights into the robustness and applicability of MRs, particularly for testing edge cases and basic behaviours.

Potential threats to external validity pertain to the specific choice of GMRs and the fact that we applied our method to only one real-world SUT. Since the assessment of the strength of an MR exclusively relies on the chosen TD and the generated mutants, we do not see any reason to assume that the proposed method is not generalisable. The fact that our proposed method worked equally well for the expert-generated HMRs and the selected SUT-relevant GMRs seems to support our belief that the specific nature of a chosen MR does not influence the applicability of our method.

7.6. Lessons Learned from the Industrial Case Study

In this contribution, we initially tested the SUT using the MT approach with HMRs. Subsequently, we proceeded with the strength assessment approach, incorporating and sharing insights provided by the industry partners. Throughout the project, several discussions were held with the industry partner to gather feedback on the application of MT in their context during the testing of SUT using MT with HMRs. These discussions culminated in a final meeting with two key representatives from the company: the main developer of the optimization code and the individual responsible for the quality of the software stack where the optimisation code is deployed.

The MT approach was deemed valuable by the developer for systematically analysing the system's limits. The discussions on generally valid HMRs provided clarity on the condition space of the observed code, even for experienced developers. This process not only enhanced the developer's understanding of the SUT but also highlighted potential edge cases that were previously overlooked. Additionally, MT was confirmed to be a feasible strategy for ensuring the robustness of the SUT.

A significant positive side-effect was that the topic of testing was approached from a more abstract perspective. By focusing on input-output relations rather than specific outputs, the MT approach encouraged a deeper consideration of invariant properties and behavioural patterns in the SUT. This shift in perspective was appreciated as it provided a fresh way to think about testing in complex sys-

tems. The added value of the MT approach for the specific use case was assessed differently by the stakeholders. On one hand, the function chosen for testing was of moderate complexity but relatively straightforward to understand. The underlying codebase was small, involving only one additional external file. On the other hand, the MT approach demonstrated its potential even in this simpler scenario, reinforcing its relevance for testing more challenging cases. These include systems that (a) are difficult to test in isolation, (b) cannot be easily divided into small functional units, and (c) are subject to frequent modifications.

7.7. Conclusions and Future Directions

MT and mutation testing are both techniques used to uncover errors in systems when there is a lack of a reliable test oracle or when creating one is not feasible. In mutation testing, there's an underlying assumption that the original program is correct. However, this is not always a safe assumption. In contrast, MT does not rely on this premise; Once the MR are defined, MT provides a structured approach to verify these rules. The reliability of the system can then be inferred based on the adherence to these MRs. The efficacy of MR varies, and it's important to rank these rules based on their bug-finding capabilities.

This study advances the understanding and application of MT by introducing a structured approach to assess MR effectiveness through the fusion of MetaTrimmer—an approach for selecting and constraining MRs using TD—and traditional mutation testing. We starts by evaluating the applicability of each MR based on TD with MetaTrimmer (*Phase 1 - Applying MetaTrimmer*). This is complemented by mutation testing (*Phase 2 - Mutation Testing*), responsible for generating mutated versions of the SUT, verifying their executability, and performing initial analyses to determine the fate of mutants (killed, survived, or equivalent). Subsequently, we use MetaTrimmer with the mutated SUT versions and compare them with the rates of killed versus survived from mutation testing analysis (*Phase 3 - Assessing the Effectiveness of HMRs and GMRs*). To comprehensively assess MR effectiveness, we employ a three-level strategy to compare outcomes with those obtained from mutation testing.

The first level of our evaluation focused on MRs' ability to detect equivalent mutants, which do not alter the program's behaviour in any observable way. This highlighted the inherent challenge MRs encounter in pinpointing such subtle variations. It's crucial to recognise that expecting MRs to detect these mutants with ease may not be realistic. The second level contrasted these findings by testing MRs against mutants that were consistently detected ('killed') by all TD inputs. This provided insight into the MRs' capability to catch significant errors, comparing favourably to traditional mutation testing approaches. In this scenario, effective MRs should correspondingly trigger violations at least for one TD input. The third level dealt with 'mixed cases', where mutants showed variable responses to different TD inputs. This final assessment layer was essential for quantifying the

TD necessary to trigger violations, thus measuring the sensitivity and comprehensive detection capability of each MR. Instances where MRs triggered violations, despite mutants surviving certain TD inputs, emphasise the need for MRs to be evaluated not just on their fault detection rate but also on their sensitivity to TD.

Our study provides deep insights into the effectiveness of MRs. First, detecting equivalent mutants remains a significant challenge within mutation testing and MT. Furthermore, the effectiveness of MRs is clearly demonstrated when all TD consistently kills mutants. In such scenarios, an effective MR detects mutants by the triggering of violations across at least one instance of TD. In addition to effectiveness, the sensitivity of MRs is paramount. Sensitivity is linked to the amount of TD needed to trigger a violation. Our study's investigation into 'mixed cases' uncovers scenarios where specific TD fail to detect mutations, yet MRs detect violations. This underscores the importance of assessing MRs based on their sensitivity to variations across diverse TD inputs. A highly sensitive MR operates effectively across a broader range of inputs, making it a desirable attribute when testing a system.

Regarding the future directions, this study has opened the pathway for deeper investigation into the 'softness' of MRs, a concept that calls for a balanced tolerance. Future research should focus on quantifying the degree of softness that is both acceptable and effective in various contexts, potentially through a systematic classification of MRs based on their tolerance. Such work could explore how softness impacts the detection of near-miss faults—errors that only marginally affect the program's output. This could be done by systematically analysing the cost-benefit trade-off involved in implementing 'softer' MRs and their impact on the overall reliability and accuracy of the MT process.

Beyond their use in detecting faults, MRs possess inherent debugging capabilities, each providing insights upon violation. The nature of these insights is directly tied to the properties of the SUT they are designed to test. For example, in our study, the violation of HMR_{2-PER} is showed in only 4 out of 405 mutated programs; when inspecting the mutants and the violation, it pointed to a sorting function. This precise correlation suggests that violations of MRs can serve as indicators for potential areas of concern within the code. Future work should leverage this diagnostic aspect of MRs. By constructing MRs that relate closely to the structural elements of the SUT, such as key functions or algorithms, researchers could use MR violations to not only detect the presence of faults but also narrow down their probable location within the codebase.

7.8. Replication Package

Due to confidentiality agreements, we cannot share the SUT's source code or mutated versions. Instead, we offer template scripts for replicating our method with any MATLAB program. The complete data from our experiments is available on our GitHub repository: Assessing-the-Strength-of-Metamorphic-Testing Repo.

8. DISCUSSION

In this chapter, we discuss the key contributions of the thesis and evaluate their implications. This discussion highlights the strengths of the contribution of the thesis and examines their limitations. We begin by revisiting the Contribution No. 1 - *a TD-driven classification method* in Section 8.1, followed by the Contribution No. 2 - *a method for refraining MRs by settings constrains based on TD* in Section 8.2, and finally, the Contribution No. 3 - *a method for assessing the strengths of MRs* in Section 8.3.

8.1. Method for Classifying MTs Based on TD

We presented MetaTrimmer, a TD-driven approach for classifying MRs based on TD. MetaTrimmer marks a significant shift from traditional static, code-structure-based MR classification methods to a dynamic, TD-driven approach. By evaluating the behaviour of MRs across different TD inputs, MetaTrimmer provides a more flexible and context-aware classification system. Moreover, it enables the extraction of constraints that help narrow the scope of MR applicability, allowing testers to craft more targeted and efficient test cases by ensuring comprehensive testing coverage. This dual functionality improves both the precision of MR classification and the overall test coverage.

While further validation is needed across more varied software systems, these results indicate that MetaTrimmer effectively enhances MR classification by incorporating TD behaviour into the decision-making process. It is important to note that MetaTrimmer is designed to automate the process of classifying MRs based on TD, which offers significant time and effort savings compared to manual classification. Additionally, MetaTrimmer's methodology is domain-agnostic, allowing its application to diverse software systems and domains. This practical relevance makes it a versatile solution suitable for various testing scenarios, regardless of the source code or specific domain.

While MetaTrimmer provides a more robust classification system, it still relies on manual intervention during the MR analysis process, which becomes more challenging when dealing with large datasets. Another potential limitation lies in the dependency on the diversity and representatives of the generated TD. Since MetaTrimmer classifies MRs based on TD behaviour, the accuracy and comprehensiveness of its classifications are only as good as the TD used. If the TD does not adequately cover the input space or relevant edge cases, MRs that should be classified as violated or always not violated may be incorrectly placed into mixed, leading to less effective test suites. MetaTrimmer currently operates within a specific SUT context (numerical programs), and its generalisability across diverse domains remains to be fully validated.

8.2. Method for Refraining MRs by Settings Constrains Based on TD

We presented an enhancement of MetaTrimmer, specifically targeting the MR Analysis step, to refine MRs that fall into the mixed cases by identifying constraints within the TD space. This enhancement defines when an MR behaves as always not violated and when it behaves as always violated. By doing so, it enhances the usability of mixed MRs, allowing them to function as both positive (not violated) and negative (violated) test cases. This process not only expands the test suite but also improves its effectiveness by providing clearer insights into the applicability of mixed MRs.

The proposed enhancement, MetaTrimmer+, combines manual inspection of test logs with ARM. ARM plays a crucial role in automating the extraction of patterns from TD, once the TD is described in a structured form. This automation helps overcome the challenges of manually defining constraints for MRs, streamlining the process and improving overall efficiency in refining MRs.

While the MetaTrimmer+ approach shows promising results, further validation is needed to fully assess its robustness. A key limitation lies in its dependency on the accuracy and comprehensiveness of the patterns extracted by ARM. Although ARM helps automate the identification of constraints for mixed MRs, it may not always capture the full complexity of TD patterns, especially in cases involving non-linear relations or highly intricate system behaviours. In such scenarios, ARM could produce conflicting or incomplete rules, necessitating significant manual intervention to resolve ambiguities.

Moreover, the effectiveness of the ARM-based approach is closely tied to how well the TD is described. The process of converting TD into a set of descriptive characteristics is critical for generating meaningful patterns. If the chosen descriptors fail to capture essential properties of the TD, or if they are overly simplistic, the resulting rules may not accurately reflect the behaviour of the MRs. For instance, while MetaTrimmer+ utilises descriptors like list size, presence of zeros, and minimum/maximum values, these may not be adequate for more complex input types or behaviours. The richness and relevance of the descriptors directly affect ARM's ability to generate useful patterns. If key features are overlooked, the constraints identified for the mixed cases could be incomplete or misleading, limiting the method's effectiveness in more intricate or domain-specific scenarios. Thus, the overall success of the approach is highly dependent on the quality and depth of the descriptors used.

8.3. Assessing the Strengths of MRs

We introduced a method to assess the defect-detection strengths of test suites generated from MRs, offering a strategy to rank MRs by their bug-finding capabilities through a combination of MetaTrimmer and mutation testing. This approach allows testers to prioritise the most effective MRs, optimising the test suite by reducing its size without sacrificing defect-detection capacity. By focusing on the most untactful MRs, the method ensures a lean yet highly efficient test suite. Through the evaluation of both HMRs and GMRs, the approach effectively identified the MRs that produced the most valuable test cases. Its successful application in an industrial case study with a real-world SUT from an Austrian company further demonstrates its practical relevance and applicability.

However, the primary limitation of this method is the computational overhead associated with mutation testing, particularly when applied to large and complex SUTs. Additionally, the method's success is contingent on the quality and diversity of the TD. If the TD lacks coverage or does not reflect the full range of possible system behaviours, the method's ability to detect defects could be diminished, potentially affecting the accuracy of the MR rankings.

8.4. Threats to Validity

In this section, we discuss the threats to the validity of the thesis as a whole, covering the primary contributions and overarching findings. These threats are categorised into construct, internal, external and conclusion validity.

8.4.1. Construct Validity

Construct validity relates to the appropriateness of the constructs and approaches proposed in this thesis, including MetaTrimmer, MetaTrimmer+, and the approach for assessing the strength and ranking of MRs. A significant challenge for all these approaches is their reliance on the adequacy and diversity of the TD. These approaches are inherently data-intensive, meaning their effectiveness is strongly influenced by the quality, diversity, and representativeness of the TD. For example, in the MetaTrimmer and MetaTrimmer+ approaches, insufficient diversity in the TD may fail to trigger mixed cases, which are a key aspect of the method's novelty. Mixed cases arise when certain MRs exhibit inconsistent behaviours depending on the characteristics of the TD. If the TD lacks sufficient variation to cover edge cases or critical behaviours of the SUT, mixed cases may not emerge, and the analysis may underestimate the applicability of the proposed approach.

Similarly, the assessment of MR strength and the ranking mechanism depend heavily on the ability of the TD to elicit diverse behaviours in the SUT. If the input data does not adequately exercise the system's functionality, the resulting evaluation may fail to accurately reflect the true capabilities of the MRs or the SUT's behaviour under varied conditions. The same holds for mutation testing,

where the detection of equivalent and trivial mutants, as well as the identification of level 3 mutants for ranking, is intrinsically linked to the comprehensiveness of the TD.

To mitigate this threat, we ensured that the TD used during the evaluation of MetaTrimmer and MetaTrimmer+ included a diverse range of types. This diversity aimed to encompass various input characteristics and behaviours, increasing the likelihood of observing mixed cases. However, we acknowledge that the evaluation was conducted on methods that are relatively simple in nature and with TD that lacks high complexity. While this simplifies the experimental setup and enables clearer analysis, it may not fully capture the challenges associated with more complex, real-world systems. The evaluation conducted with the industrial partner addresses this limitation by reusing TD that typically is used by the industrial partner. While the evaluation of the approaches for classifying, refining, and ranking MRs presented in this thesis provides meaningful insights, extending it to include more complex methods and diverse datasets from different domains would further strengthen the generalisability of the findings and demonstrate their broader applicability.

8.4.2. Internal Validity

Internal validity relates to the soundness of the methodologies used to evaluate the hypotheses and contributions of the thesis. A significant threat to internal validity is the adequacy and diversity of the TD used in our experiments. As explained previously, since the effectiveness of MRs and their evaluation is inherently tied to the TD, any limitations in the TD could skew the results.

Another potential threat to internal validity is with regard to the MRs used in the thesis. While the primary goal of the first two contributions, MetaTrimmer and MetaTrimmer+, was to demonstrate the capability of the proposed approaches to select and refine MRs from a predefined pool, the size and diversity of the MR pool itself may introduce limitations. Specifically, the predefined set of six MRs, while adequate for controlled evaluations and comparisons with the PMR approach, may not represent a sufficiently large or diverse set to fully assess the scalability and robustness of the proposed methods. To address this, the third contribution expanded the evaluation by including both HMRs and GMRs. HMRs were derived from domain expertise, ensuring their alignment with the SUT's characteristics and industrial context. GMRs were selected from well-documented transformations in the MT literature, providing a broader set of MRs applicable across various systems. This combined approach mitigated the limitations of relying solely on predefined MRs, allowing for a more comprehensive assessment of the proposed methods.

Another potential threat to internal validity in the assessment of MR strength is whether the inclusion of equivalent and trivial mutants (typically discarded in mutation testing) accurately reflects the true strength of MRs. While these mutants

are leveraged to identify particularly strong or weak MRs (levels 1 and 2), there is a risk that relying on these constructs might not fully capture all relevant aspects of MR behaviour. In this thesis, this threat is mitigated by carefully designing the evaluation process to ensure that equivalent and trivial mutants are used exclusively to highlight extremes in MR strength. Furthermore, the inclusion of level 3 mutants, which represent more the sensitivity of MRs to varying TD characteristics, provides a robust mechanism for ranking MRs beyond the limitations of equivalent and trivial mutants. By combining these constructs, we reduce potential biases in the assessment process and ensure that the rankings accurately reflect the real-world effectiveness of MRs.

8.4.3. External Validity

External validity concerns the generalizability of the findings and methods to other systems, domains, and contexts. A significant threat to external validity pertains to the existence of mixed cases in real-world scenarios. Mixed cases, introduced as a novel concept, represent situations where the non-violation/violation of a MR depends on specific characteristics of the TD and the behaviour of the SUT. The frequency of mixed cases in practice is influenced by the diversity of the TD, the complexity of the SUT, and the applicability of the MRs to the testing context.

Ideally, the proposed approaches for classifying, refining, and ranking MRs are intended for early stage of creating a regression test suite. This aligns with the exploratory testing paradigm, where the primary goal is to uncover unexpected behaviours or inconsistencies in the SUT based on diverse and exploratory TD. In such scenarios, mixed cases are more likely to occur due to the absence of strict predefined expectations and the reliance on input-output relations defined by the MRs. In the exploratory phase, TD is often generated with high variability to cover different parts of the SUT, which increases the likelihood of encountering input subsets that trigger mixed cases.

While the evaluation in this thesis confirmed the existence of mixed cases in controlled experiments, real-world systems often exhibit greater complexity and variability, suggesting that mixed cases may be even more prevalent in practical applications. Future studies focusing on diverse domains and systems could provide further evidence to support this hypothesis.

Another potential threat to external validity pertains to the specific choice of MRs and the fact that the proposed methods were evaluated on relatively simple SUTs and only one real-world SUT. Since MetaTrimmer, MetaTrimmer+, and the assessment of MR strength rely exclusively on the chosen TD and, in the case of strength assessment, the generated mutants, there is no reason to assume that the proposed methods are not generalizable. The fact that the proposed methods worked equally well with expert-generated HMRs and the selected SUT-relevant GMRs supports the belief that the specific nature of a chosen MR does not influence the applicability of the methods.

8.4.4. Conclusion Validity

Conclusion validity pertains to the reliability of the relationships and patterns identified in the study. The conclusions drawn in this thesis are based on empirical data, statistical analyses, and results observed from experiments conducted with a real-world SUT. A potential threat to conclusion validity arises from the limited diversity of experiments. While the results demonstrate clear relationships between TD characteristics and the effectiveness of MRs, as well as their classification and refinement, it is possible that different systems, TD, or experimental setups could yield varying results. To mitigate this threat, the reproducibility of the experiments has been ensured by sharing all generated data and scripts used during the evaluations conducted across the contributions. This transparency allows others to replicate the findings and validate the results in different contexts, thereby strengthening the reliability of the conclusions

9. CONCLUSION

In the research conducted in this thesis, the focus has been on advancing the field of MT, particularly by developing new methods that enhance the process of generating, classifying, refining, and assessing MR.

Traditionally, the logic behind MR selection using binary classifiers assumes that a chosen MR must universally apply across the entire valid input data space. However, this assumption is often unrealistic, as MRs may only apply to specific subsets of the input space. As demonstrated in this thesis, relying on the belief that an MR must consistently apply to all valid input data can lead to false positives, where a violation is incorrectly perceived as a fault, even though there may be none. This thesis challenges the assumption of universal applicability by introducing TD-driven methods to classify and refine MRs.

The first contribution of this thesis, MetaTrimmer, introduces a TD-driven method for classifying MRs. By evaluating the behaviour of MRs across different TD inputs, MetaTrimmer shifts away from traditional static, code-structure-based classification methods toward a more dynamic, context-aware system. MetaTrimmer offers a more flexible and precise way to classify MRs, accounting for cases where MRs may only apply to specific TD subsets. MetaTrimmer's ability to extract constraints that narrow the scope of MR applicability further enhances its utility, allowing testers to create more targeted and effective test cases. By incorporating TD into the decision-making process, MetaTrimmer addresses the limitations of binary classifiers and offers a more robust solution to MR classification.

While MetaTrimmer has shown promise in classifying MRs based on TD, its current application is limited to specific types of SUTs. A key future direction is expanding MetaTrimmer's applicability across diverse software domains, including complex systems like AI and ML models, where the behaviour of MRs may vary greatly across different input data spaces. Additionally, future research could explore integrating MetaTrimmer with other test generation techniques, such as fuzz testing or model-based testing, to enhance the diversity of generated TD and ensure better coverage of edge cases.

The second contribution, extends the MR analysis of MetaTrimmer, by refining MRs that fall into the mixed-cases category. This enhancement helps identify constraints within the TD space, defining when an MR behaves as always not violated or always violated. By combining manual inspection of test logs with automated pattern extraction using ARM, MetaTrimmer+ enhances the usability of mixed MRs. These MRs can now function as both positive and negative test cases, improving the overall effectiveness of the test suite. However, the success of this approach heavily depends on the quality of the TD descriptions and the ability of ARM to capture the full complexity of TD patterns. Despite these challenges, MetaTrimmer+ represents a significant step forward in refining MRs.

MetaTrimmer+ successfully introduces an ARM-based approach for refining the MRs that present mixed cases, but its effectiveness hinges on the quality of the TD descriptors used. Future work could focus on enhancing the descriptor generation process, particularly for more complex or domain-specific systems, by incorporating richer and more context-aware features. In addition, leveraging deep learning techniques for feature extraction could improve the identification of non-linear relations within TD, further refining the accuracy of the constraints applied to MRs that present mixed cases. Another potential direction would be exploring hybrid approaches that combine multiple pattern extraction methods to complement ARM, thereby increasing the reliability of the refinement process when dealing with intricate system behaviours or non-traditional input data types. Beyond ARM, alternative approaches for defining constraints could also be explored. For instance, graph-based techniques use nodes and edges to represent relations between TD features and MR behaviours. Bayesian networks [95], on the other hand, provide a probabilistic framework that can model uncertainty in MR behaviour and define constraints based on conditional probabilities derived from the TD [96]. Additionally, LLMs could be explored to find patterns and provide constraints by leveraging their ability to learn complex relations between the violation status of MRs and the TD.

Constraint programming [82] or optimization-based approaches [91] could formalize constraints as solvable mathematical relations, providing a systematic and rigorous way to evaluate and refine MRs. Additionally, rule induction methods [97], such as decision trees or rule-based algorithms, could generate interpretable constraints by extracting logical rules from the TD, offering insights into the behaviour of the SUT that align with the MRs. While these alternative approaches offer promising opportunities for defining constraints, they also come with challenges, such as scalability limitations, computational overhead, and the need for domain-specific customization to address complex or specialized systems.

Finally, the third contribution is a method for assessing the strengths of MRs based on their defect detection capabilities. By integrating MetaTrimmer with mutation testing, this approach ranks MRs by their bug-finding effectiveness, allowing testers to focus on the most valuable MRs and optimise the test suite by reducing its size without sacrificing defect-detection capacity. This method was successfully applied in an industrial case study, demonstrating its practical relevance. While the approach offers clear advantages, such as the ability to prioritise the most effective MRs, it also faces limitations in terms of computational overhead, particularly for large and complex systems. Additionally, the success of the method depends on the quality and diversity of the TD used, as inadequate TD coverage may impact the accuracy of MR rankings.

Future research could explore optimising mutation testing by developing more efficient mutant selection techniques, such as using ML to predict which mutants are most likely to reveal faults. Additionally, the study made in the contribution three has opened the pathway for deeper investigation into the 'softness' of MRs,

a concept that calls for a balanced tolerance. Future research should focus on quantifying the degree of softness that is both acceptable and effective in various contexts, potentially through a systematic classification of MRs based on their tolerance. Such work could explore how softness impacts the detection of near-miss faults—errors that only marginally affect the program's output. This could be done by systematically analysing the cost-benefit trade-off involved in implementing 'softer' MRs and their impact on the overall reliability and accuracy of the MT process.

Beyond their use in detecting faults, MRs possess inherent debugging capabilities, each providing insights upon violation. The nature of these insights is directly tied to the properties of the SUT they are designed to test. Future work should leverage this diagnostic aspect of MRs. By constructing MRs that relate closely to the structural elements of the SUT, such as key functions or algorithms, researchers could use MR violations to not only detect the presence of faults but also narrow down their probable location within the codebase.

BIBLIOGRAPHY

[1] T. M. Abdellatif, L. F. Capretz, and D. Ho, "Software analytics to software practice: A systematic literature review," in *2015 IEEE/ACM 1st Int'l Workshop on Big Data Software Engineering*, 2015, pp. 30–36.

[2] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, and M. Souza, "A machine learning approach to generate test oracles," in *Proc. of the XXXII Brazilian Symp. on Softw. Eng.*, ser. SBES '18, Sao Carlos, Brazil: Association for Computing Machinery, 2018, pp. 142–151.

[3] K. Patel and R. M. Hierons, "A partial oracle for uniformity statistics," *Softw. Quality Journal*, vol. 27, no. 4, pp. 1419–1447, 2019.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. on Softw. Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[5] 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 (QuASoQ@APSEC)*, vol. 2767, 2020, pp. 21–28.

[6] 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.

[7] A. Duque-Torres, D. Pfahl, C. Klammer, and S. Fisher, "Using source code metrics for predicting metamorphic relations at method level," in *5th Workshop on Validation, Analysis and Evolution of Software Tests*, ser. VST'22, 2022.

[8] 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.

[9] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Communications of the ACM*, vol. 62, no. 3, pp. 61–67, Feb. 2019.

[10] A. Duque-Torres, C. Klammer, S. Fischer, and D. Pfahl, "Is it the best solution? testing an optimisation algorithm with metamorphic testing," in *Product-Focused Software Process Improvement (PROFES)*, Springer Nature Switzerland, 2023, pp. 339–354.

[11] S. Yoo, "Metamorphic testing of stochastic optimisation," in *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 192–201.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] S. Segura, J. A. Parejo, J. Troya, and A. R. Cortés, "Metamorphic testing of restful web apis," *IEEE Trans. Software Eng.*, vol. 44, no. 11, pp. 1083–1099, 2018.

[17] G. Luo, X. Zheng, H. Liu, R. Xu, D. Nagumothu, R. Janapareddi, E. Zhuang, and X. Liu, "Verification of microservices using metamorphic testing," in *Algorithms and Architectures for Parallel Processing*, S. Wen, A. Zomaya, and L. T. Yang, Eds., Cham: Springer International Publishing, 2020, pp. 138–152.

[18] J. Ayerdi, P. Valle, S. Segura, A. Arrieta, G. Sagardui, and M. Arratibel, "Performance-driven metamorphic testing of cyber-physical systems," *IEEE Transactions on Reliability*, vol. 72, no. 2, pp. 827–845, 2023.

[19] J. Ayerdi, V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel, "Evolutionary generation of metamorphic relations for cyber-physical systems," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '22, Boston, Massachusetts: Association for Computing Machinery, 2022, pp. 15–16.

[20] F. Azimian, F. Faghih, M. Kargahi, and S. M. Mahdi Mirdehghan, "Energy metamorphic testing for android applications," in *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, 2019, pp. 1–6.

[21] 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.

[22] K. Rahman and U. Kanewala, "Predicting metamorphic relations for matrix calculation programs," in *3rd IEEE/ACM International Workshop on Metamorphic Testing (MET)*, ser. MET'18, 2018, pp. 10–13.

[23] 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.

[24] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "Memo: Automatically identifying metamorphic relations in javadoc comments for test automation," *Journal of Systems and Software*, vol. 181, p. 111 041, 2021.

[25] A. Duque-Torres and D. Pfahl, "Inferring metamorphic relations from javadocs: A deep dive into the memo approach," in *Product-Focused Software Process Improvement*, Springer International Publishing, 2022, pp. 418–432.

[26] Y. Zhang, D. Towey, and M. Pike, "Automated metamorphic-relation generation with chatgpt: An experience report," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023, pp. 1780–1785.

[27] Q.-H. Luu, H. Liu, and T. Y. Chen, "Can chatgpt advance software testing intelligence? a case study on metamorphic testing," *SSRN*, 2024, Available at SSRN: `https://ssrn.com/abstract=4493849` or `http://dx.doi.org/10.2139/ssrn.4493849`.

[28] S. Y. Shin, F. Pastore, D. Bianculli, and A. Baicoianu, "Towards generating executable metamorphic relations using large language models," in *Quality of Information and Communications Technology*, A. Bertolino, J. Pascoal Faria, P. Lago, and L. Semini, Eds., Cham: Springer Nature Switzerland, 2024, pp. 126–141.

[29] 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*, vol. 26, no. 3, pp. 245–269, 2016.

[30] 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, pp. 1–10.

[31] B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *3rd IEEE/ACM International Workshop on Metamorphic Testing*, ser. MET'18, 2018, pp. 14–17.

[32] 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.

[33] M. Srinivasan and U. Kanewala, "Metamorphic relation prioritization for effective regression testing," *Software Testing, Verification and Reliability*, vol. 32, no. 3, e1807, 2022.

[34] 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.

[35] A. Duque-Torres, D. Pfahl, K. Claus, and R. Ramler, "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. 1–11.

[36] A. Duque-Torres, D. Pfahl, C. Klammer, and S. Fischer, "Bug or not bug? analysing the reasons behind metamorphic relation violations," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 905–912.

[37] C. Z. Xiaowei Yan and S. Zhang, "Confidence metrics for association rule mining," *Applied Artificial Intelligence*, vol. 23, pp. 713–737, 2009.

[38] M. Asrafi, H. Liu, and F.-C. Kuo, "On testing effectiveness of metamorphic relations: A case study," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, 2011, pp. 147–156.

[39] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008)*, IEEE, 2008, pp. 94–98.

[40] F. Jafari and A. Nadeem, "Measuring effectiveness of metamorphic relations for image processing using mutation testing," *Journal of Imaging*, vol. 10, no. 4, 2024.

[41] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022.

[42] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," in *Advances in Computers*, ser. Advances in Computers, A. M. Memon, Ed., vol. 112, Elsevier, 2019, pp. 275–378.

[43] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," in *IEEE Transactions on Software Engineering*, vol. 37, Sep. 2011, pp. 649–678.

[44] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," in *Advances in Computers*, vol. 112, Elsevier, 2019, pp. 275–378.

[45] A. Duque-Torres, N. Doliashvili, D. Pfahl, and R. Ramler, "Predicting survived and killed mutants," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 274–283.

[46] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher, "Mutation-based test case generation for simulink models," in *Formal Methods for Components and Objects*, Berlin, Heidelberg, 2010, pp. 208–227.

[47] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*, Timisoara, Romania, Sep. 2010, pp. 1–10.

[48] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Proceedings of 27th International Conference on Software Engineering*, ser. ICSE'05, Saint Louis, MO, USA, USA, May 2005, pp. 402–411.

[49] M. Papadakis, D. Shin, S. Yoo, and D. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *IEEE/ACM 40th International Conference on Software Engineering*, ser. ICSE, Gothenburg, Sweden, May 2018, pp. 537–548.

[50] Y.-M. Choi and D.-J. Lim, "Model-based test suite generation using mutation analysis for fault localization," in *Applied Sciences*, vol. 9, 2019, p. 17.

[51] N. Gupta, A. Sharma, and M. K. Pachariya, "A novel approach for mutant diversity-based fault localization: Dam-fl," in *International Journal of Computers and Applications*, vol. 0, 2019, pp. 1–10.

[52] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: A mutation system for java," in *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, pp. 827–830.

[53] 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*, ser. ISSTA'14, San Jose, CA, USA, 2014, pp. 433–436.

[54] 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*, ACM, 2016, pp. 449–452.

[55] Mutatest Developers, *Mutatest Documentation*, `https://mutatest.readthedocs.io/en/latest/`, 2022.

[56] MutPy Developers, *MutPy - Mutation Testing for Python*, `https://pypi.org/project/MutPy/`, 2019.

[57] D. Hook and D. Kelly, "Mutation sensitivity testing," *Computing in Science & Engineering*, vol. 11, no. 6, pp. 40–47, 2009.

[58] J. Kisaakye, O. Kilincceker, and S. Demeyer, "Towards mutation testing of simulink models.," in *BENEVOL*, 2021.

[59] H. Runge, *A mutation analysis framework for simulink models*, 2018.

[60] J. Možucha and B. Rossi, "Is mutation testing ready to be adopted industry-wide?" In *International Conference on Product-Focused Software Process Improvement*, Springer, 2016, pp. 217–232.

[61] R. Ramler, T. Wetzlmaier, and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17, Marrakech, Morocco, 2017, pp. 1401–1408.

[62] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10, Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224.

[63] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Learning Theory and Kernel Machines*, Springer Berlin Heidelberg, 2003, pp. 129–143.

[64] A. Ben-Hur and J. Weston, "A user's guide to support vector machines," in *Data Mining Techniques for the Life Sciences*. Humana Press, 2010, pp. 223–239.

[65] U. Kanewala, "Techniques for automatic detection of metamorphic relations," in *IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2014, pp. 237–238.

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

[67] *Colt project*, `http://acs.lbl.gov/software/colt/`, Accessed: 2021-09-21.

[68] *Apache mahout*, `https://mahout.apache.org/`, Accessed: 2021-09-21.

[69] *Apache commons mathematic*, `http://commons.apache.org/proper/commons-math/`, Accessed: 2021-09-21.

[70] *Java collections*, `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`, Accessed: 2021-09-21.

[71] X. Lin, M. Simon, and N. Niu, "Exploratory metamorphic testing for scientific software," *Computing in Science Engineering*, vol. 22, no. 2, pp. 78–87, 2020.

[72] J. Brown, Z. Q. Zhou, and Y. Chow, "Metamorphic testing of navigation software: A pilot study with google maps," in *51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018*, T. Bui, Ed., ScholarSpace / AIS Electronic Library (AISeL), 2018, pp. 1–10.

[73] J. Ayerdi, A. Arrieta, E. B. Pobee, and M. Arratibel, "Multi-objective metamorphic test case selection: An industrial case study (practical experience report)," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 541–552.

[74] G. Sudheerbabu, T. Ahmad, D. Truscan, and J. Vain, "Metamorphic testing for verification and fault localization in industrial control systems," in *CyberSecurity in a DevOps Environment : From Requirements to Monitoring*, A. Sadovykh, D. Truscan, W. Mallouli, A. R. Cavalli, C. Seceleanu, and A. Bagnato, Eds. Cham: Springer Nature Switzerland, 2024, pp. 127–159.

[75] T. Y. Chen, D. Huang, T. Tse, and Z. Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, Citeseer, 2004, pp. 569–583.

[76] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *12th International Conference on Quality Software*, 2012, pp. 59–68.

[77] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'14, Vasteras, Sweden, 2014, pp. 701–712.

[78] T. Y. Chen, P.-L. Poon, and X. Xie, "METRIC: METamorphic Relation Identification based on the Category-choice framework," *Journal of Systems and Software*, vol. 116, pp. 177–190, 2016.

[79] C.-A. Sun, A. Fu, P.-L. Poon, X. Xie, H. Liu, and T. Y. Chen, "METRIC+: A metamorphic relation identification technique based on input plus output domains," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1764–1785, 2021.

[80] 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*, ser. MaLTeSQuE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 1–6.

[81] 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*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 420–424.

[82] M. C. de Castro-Cabrera, A. García-Domínguez, and I. Medina-Bulo, "Using constraint solvers to support metamorphic testing," in *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, 2019, pp. 32–39.

[83] T. Jameel, L. Mengxiang, and L. Chao, "Automatic test oracle for image processing applications using support vector machines," in *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2015, pp. 1110–1113.

[84] J. C. Carver, "Towards reporting guidelines for experimental replications: A proposal," in *1st international workshop on replication in empirical software engineering*, Citeseer, vol. 1, 2010, pp. 1–4.

[85] M. LLC. "Artifact review and badging - current." (2020), [Online]. Available: `https://www.acm.org/publications/policies/artifact-review-and-badging-current` (visited on 09/21/2021).

[86] 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.

[87] A. Duque-Torres, D. Pfahl, C. Klammer, and S. Fischer, "Exploring a test data-driven method for selecting and constraining metamorphic relations," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2023, pp. 370–377.

[88] A. P. Engelbrecht, "Appendix a: Optimization theory," in *Computational Intelligence*. John Wiley & Sons, Ltd, 2007, pp. 551–579.

[89] V. Beiranvand, W. Hare, and Y. Lucet, "Best practices for comparing optimization algorithms," *Optimization and Engineering*, vol. 18, no. 4, pp. 815–848, 2017.

[90] Q. Huang, J. Mao, and Y. Liu, "An improved grid search algorithm of svr parameters optimization," in *2012 IEEE 14th International Conference on Communication Technology*, 2012, pp. 1022–1026.

[91] M. F. Shlesinger, "Random searching," *Journal of Physics A: Mathematical and Theoretical*, vol. 42, no. 43, p. 434 001, Oct. 2009.

[92] H. M. Torun, M. Swaminathan, A. Kavungal Davis, and M. L. F. Bellaredj, "A global bayesian optimization algorithm and its application to integrated system design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 4, pp. 792–802, 2018.

[93] L. J. Perreault, M. Thornton, R. Goodman, and J. W. Sheppard, "A swarm-based approach to learning phase-type distributions for continuous time bayesian networks," in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1860–1867.

[94] 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.

[95] W. Wiegerinck, B. Kappen, and W. Burgers, "Bayesian networks for expert systems: Theory and practical applications," in *Interactive Collaborative Information Systems*, R. Babuška and F. C. A. Groen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 547–578.

[96] V. O. Barth, H. O. Caetano, C. D. Maciel, and M. Aiello, "Quantifying uncertainty in bayesian networks structural learning," in *2024 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, 2024, pp. 1–8.

[97] J. W. Grzymala-Busse, "Rule induction, missing attribute values, and discretization," in *Granular, Fuzzy, and Soft Computing*, T.-Y. Lin, C.-J. Liau, and J. Kacprzyk, Eds. New York, NY: Springer US, 2023, pp. 389–399.

[98] *Pyml toolkit*, `http://pyml.sourceforge.net/`, Accessed: 2021-09-21.

[99] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," in *Journal of Machine Learning Research*, vol. 12, 2011, pp. 2825–2830.

[100] S. Cass, "The top programming languages: Our latest rankings put python on top-again-[careers]," *IEEE Spectrum*, vol. 57, no. 8, pp. 22–22, 2020.

[101] *Pycfg*, `https://pypi.org/project/pycfg/`, Accessed: 2021-09-21.

[102] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler, "Static race detection for device drivers: The goblint approach," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE'16, ACM, 2016, pp. 391–402.

[103] *Goblint github*, `https://github.com/goblint/analyzer`, Accessed: 2021-09-21.

[104] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *2009 International Forum on Computer Science-Technology and Applications*, vol. 1, 2009, pp. 131–136.

[105] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[106] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[107] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[108] A. Duque-Torres, N. Doliashvili, D. Pfahl, and R. Ramler, "Predicting survived and killed mutants," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 274–283.

[109] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 160–171.

[110] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.

Appendix A. REPLICATION STUDY ON PREDICTING METAMORPHIC RELATIONS

Appendix A provides detailed information on the replication study conducted as part of Chapter 5. Our replication aims to verify the findings, assess the generalisability of the PMR across different programming languages, and extend the approach with additional evaluations. The results and methodologies discussed here form a critical foundation for the advancements proposed in this thesis.

A.1. Replication Methodology

Our goal is to investigate whether the PMR approach of Kanewala *et al.* (i) can be replicated when using our own implementation of the pipeline for developing classifiers starting out from Java source code instead of CFG representations, (ii) classifiers trained on Java source code can be transferred to Python and C++ methods that have identical functionality, and (iii) the PMR approach can be applied to Python and C++ code when classifiers are developed from scratch in the target programming languages. Each of these scenarios gives rise to a research question that we answer in our study. In all scenarios, we follow the PMR procedure, Figure 5, and we use the same set of six pre-defined MRs used in the original study, *i.e.,* Section 3.5.2. However, we develop our own pipeline and create new datasets. For performance evaluation, we employ 10-fold stratified cross-validation., *i.e.,* the dataset is randomly partitioned into ten subgroups. The classifier is then built using nine subsets, with the 10th subset being used to evaluate the predictive model's performance. This procedure is done ten times, with each of the ten subgroups being evaluated separately. The ten folds in stratified 10-fold cross-validation are partitioned in such a way that they include about the same proportion of classes as the original data set. The resulting overall performance is measured as the average of the ten cross-validation tests.

Our replication package containing results, scripts, models and datasets is available online[1].

A.1.1. Research Questions

We aim at answering the following research questions

- **RQ_1:** [Replicability] *How well do classifiers predict matching MRs for Java methods when using our processing and training pipeline starting from source code?*
- **RQ_2:** [Transferability] *How well do classifiers developed on Java code predict matching MRs for functionally equivalent methods implemented in Python and C++?*

[1] https://github.com/aduquet/RENE-PredictingMetamorphicRelations

- **RQ_3:** [Generalisability] *How well do classifiers predict matching methods for Python and C++ methods when developed from source code in the respective target languages?*

A.1.2. RQ_1: How well do classifiers predict matching MRs for Java methods when using our pipeline implementation?

In RQ_1, we investigate the replicability of the PMR approach when starting out directly from Java source code (instead of CFG representations), performing all steps of feature extraction and re-generating the classifiers with a different ML package. In particular, we are interested in checking whether the classifiers developed by us achieve the same performance as published in [29]. Satisfactory results for RQ_1 are the pre-requisite for tackling RQ_2 and RQ_3.

To compare with the work of Kanewala *et al.*, we develop our own artefacts for *Phase II - Data preparation, Step 2.1 – Feature extraction*, and all the artefacts needed by *Phase III Training and testing*. Then we compare the results of SVM obtained by Kanewala *et al.* [29] who used *PyML Toolkit* [98] with our results achieved using Python *scikit-learn* library [99] with default parameter settings. For the comparison we use two datasets. The first dataset is the one used in the original study by Kanewala *et al.*, *i.e.,* the methods from Table 6. This dataset contains the CFG representations of the 100 Java methods in DOT format. In the following, we call this dataset DS_{JK}. The second dataset contains the Java source code of the 100 methods from Table 6 as contained in the open-source libraries. We call this dataset DS_{JV}. To construct this dataset, we manually translated the CFG into Java source code, ensuring that the functionality of each method aligns with the descriptions provided in the original CFG representations.

When using DS_{JK} we apply the PMR approach from *Phase I - Step 1.2* through *Phase III - Step 3.3*, since this dataset already contains the CFG representation of each method in DOT format. When using DS_{JV}, we apply our entire pipeline implementing the PMR approach, *i.e.,* from *Phase I - Step 1.1* to *Phase III - Step 3.3*. To get the CFG representation in DOT format for the dataset DS_{JV}, we use the *soot*[62] Java framework, configured so that the output matches the CFGs of the original dataset. Then, we compare the performance of our classifiers against the results published by Kanewala *et al.* In particular, we compare BSR and AUC since these are the measures provided in the original study [29]. In addition, we provide the performance measures detailed in Appendix B.1.2 - *Step 3.3 – Performance evaluation*.

A.1.3. RQ_2: How well do classifiers developed on Java code predict matching MRs for functionally equivalent methods implemented in Python and C++?

In RQ_2, we check whether classifiers developed with the PMR approach from Java methods achieve the same performance when applied to methods with identical functionality but implemented in Python or C++. We chose Python and C++ because both are popular and widely used programming languages supporting a broad range of applications [100].

For this experiment, we created two new datasets containing source code of methods written in Python and in C++. Each method in these datasets was manually mapped from its equivalent in the Java source code. The methods in each dataset are functionally identical to that of the 100 Java methods described in Table 6. The corresponding method implementations were either retrieved from the *NumPy* package for scientific computing in case of Python, the *Blinz++* high-performance library for scientific computing in case of C++, or they were implemented in Python/C++ by the authors if not present in these libraries. Because the functionality of the Python and C++ methods is equivalent to the functionality of the Java methods, we can assume that exactly the same MRs that match the Java methods match the corresponding Python and C++ methods. The dataset named DS_{PY} contains the Python methods, and the dataset named DS_{C++} contains the C++ methods. To get the graph representation in DOT format, Section 3.5.1: *Phase I - Step CFG generation*, for the methods written in Python, we use the Python package *pycfg* [101], and for the methods written in C++, we use *Goblint*[102], [103].

A.1.4. RQ_3: How well do classifiers predict matching methods for Python and C++ methods when developed from source code in the respective target languages?

Finally, in RQ_3, we check whether the PMR approach works for Python and C++ code similarly well as it does for Java code, if we develop the classifiers for each target language from scratch. Thus, we train SVM models using each dataset, *i.e.,* DS_{PY} and DS_{C++}. We compare the performance of the new classifiers against the results obtained in RQ_1 and RQ_2.

A.2. Results and Discussion

A.2.1. RQ_1 How well do classifiers predict matching MRs for Java methods when developed from source code using our pipeline?

Table 22 shows the performance of our PMR implementation for both Java datasets, DS_{JK} and DS_{JV}. Overall, regardless of the feature extraction technique used, the results are fairly close. This can be seen in the Error column, which displays the

difference in performance between DS_{JK} and DS_{JV} for each MR. The most negative value is -0.104 (Accuracy of ADD) while the farthest positive value is 0.148 (BSR of INV). This indicates that the classifiers developed by us are consistent for Java code independent from the starting point of the model development (CFG vs. source code). Table 23 shows how the performance of our PMR implementation compares to the performance obtained by Kanewala *et al.* in terms of AUC and BSR. As can be seen from the Error column, our results are close to those obtained in the original study. The Error range is [-0.093, 0.061] for AUC and [-0.118, 0.117] for BSR. From combining the results shown in Table 23 with those shown in Table 22 we conclude that our implementation of PMR achieves similar performance as reported in [29] even when starting out from source code.

With regards to *replicability* (RQ_1), our results indicate that we can achieve similar results as Kanewala *et al.* when re-implementing the PMR approach no matter whether we start the modelling process from source code or from CFG representations.

Table 22. PMR performance achieved by our classifiers when starting from DS_{JV} and DS_{JK}

MR	Feat$^{\perp}$	Performance measurements																	
		Accuracy			Precision			Recall			f-measure			AUC			BSR		
		DS_{JK}	DS_{JV}	Error$^{\pm}$	DS_{JK}	DS_{JV}	Error$^{\pm}$	DS_{JK}	DS_{JV}	Error$^{\pm}$	DS_{JK}	DS_{JV}	Error$^{\pm}$	DS_{JK}	DS_{JV}	Error$^{\pm}$	DS_{JK}	DS_{JV}	Error$^{\pm}$
ADD	NF-NP	**0.802**	0.787	0.015	**0.786**	0.751	0.035	**0.812**	0.704	0.108	0.773	**0.775**	**-0.002**	**0.837**	0.827	0.010	0.768	**0.785**	**-0.017**
	GK	0.712	**0.816**	**-0.104**	0.702	**0.732**	**-0.030**	0.717	**0.758**	**-0.041**	**0.744**	0.712	0.032	**0.769**	0.707	0.062	**0.737**	0.729	0.008
	RWK	0.851	**0.86**	**-0.009**	**0.836**	0.712	0.124	0.771	**0.791**	**-0.020**	**0.786**	0.785	0.001	**0.905**	0.877	0.028	**0.843**	0.829	0.014
MUL	NF-NP	**0.712**	0.688	0.024	0.672	**0.689**	**-0.017**	**0.685**	0.661	0.024	0.657	**0.705**	**-0.048**	**0.742**	0.734	0.008	0.631	**0.654**	**-0.023**
	GK	**0.663**	0.641	0.022	0.714	**0.732**	**-0.018**	0.697	**0.758**	**-0.061**	0.676	**0.733**	**-0.057**	**0.775**	0.730	0.045	**0.689**	0.657	0.032
	RWK	**0.789**	0.695	0.094	0.666	**0.706**	**-0.040**	0.693	**0.797**	**-0.104**	0.660	**0.676**	**-0.016**	**0.846**	0.820	0.026	**0.774**	0.739	0.035
PER	NF-NP	0.838	**0.840**	**-0.002**	0.860	**0.883**	**-0.023**	0.835	**0.846**	**-0.011**	**0.855**	0.790	0.065	**0.945**	0.925	0.020	**0.847**	0.813	0.034
	GK	**0.834**	0.826	0.008	**0.888**	0.819	0.069	0.823	**0.845**	**-0.022**	0.864	0.79	0.074	**0.872**	0.811	0.061	**0.853**	0.839	0.014
	RWK	0.916	**0.918**	**-0.002**	**0.917**	0.827	0.090	**0.878**	0.835	0.043	0.877	**0.893**	**-0.016**	**0.963**	0.944	0.019	0.757	**0.793**	**-0.036**
INC	PF-NP	0.792	**0.807**	**-0.015**	**0.847**	0.822	0.025	**0.837**	**0.837**	**0.000**	**0.776**	0.759	0.017	**0.845**	0.852	**-0.007**	**0.793**	0.786	0.007
	GK	0.752	**0.788**	**-0.036**	0.721	**0.781**	**-0.060**	**0.790**	0.776	0.014	**0.783**	0.718	0.065	0.850	**0.882**	**-0.032**	**0.762**	0.744	0.018
	RWK	0.799	**0.839**	**-0.040**	**0.832**	0.792	0.040	**0.800**	0.773	0.027	**0.854**	0.764	0.090	**0.862**	0.821	0.041	**0.673**	0.654	0.019
EXC	NF-NP	**0.763**	0.753	0.010	0.772	**0.783**	**-0.011**	**0.778**	0.759	0.019	0.762	**0.789**	**-0.027**	**0.768**	0.755	0.013	**0.868**	0.839	0.029
	GK	**0.816**	0.787	0.029	**0.816**	0.861	**-0.045**	0.849	**0.890**	**-0.041**	**0.871**	0.790	0.081	**0.873**	0.870	0.003	**0.758**	0.755	0.003
	RWK	**0.774**	0.725	0.049	**0.757**	0.743	0.014	**0.757**	0.741	0.016	**0.769**	0.744	0.025	**0.731**	0.727	0.004	**0.79**	0.757	0.033
INV	NF-NP	**0.714**	0.705	0.009	**0.674**	0.659	0.015	**0.702**	0.671	0.031	0.675	**0.694**	**-0.019**	0.905	**0.917**	**-0.012**	0.656	**0.661**	**-0.005**
	GK	**0.778**	0.759	0.019	0.765	**0.769**	**-0.004**	**0.738**	0.737	0.001	**0.760**	0.721	0.039	**0.671**	0.670	0.001	**0.679**	0.655	0.024
	RWK	**0.651**	0.585	0.066	0.610	**0.659**	**-0.049**	**0.643**	0.639	0.004	**0.675**	0.653	0.022	0.760	**0.766**	**-0.006**	**0.787**	0.639	0.148

$^{\perp}$Feature extraction approach, $^{\pm}$Error ($DS_{JK} - DS_{JV}$), **NF-PF:** Node Feature - Path Feature, **GK:** Graphnet Kernel, **RWK:** Random Walk Kernel

A.2.2. RQ_2 How well do classifiers developed on Java code predict matching MRs for functionally equivalent methods implemented in Python and C++?

Table 24 reports on the performance when using classifiers, developed starting out from the DS_{JV} dataset, to predict matching MRs for methods contained in the DS_{PY} and DS_{C++} datasets. The assumption behind applying a classifier built on Java code to methods that are functionally equivalent but implemented in a different programming language is that the CFG representations from which the

features in the SVM models are taken would be similar enough to achieve similar classification performance as when applied to Java methods.

However, as shown in Table 24, the performance is low for all performance measures and for both Python and C++. No measure is greater than 0.689. This result suggests that the representation of the CFGs of the Python and C++ methods to which the feature extraction algorithm is applied are more different from the CFGs of the Java methods than expected. This can be explained due to the language-specific CFG generators that we used as well as differences in the way how the methods (with identical functionality) are implemented in different programming languages.

With regards to *transferability* (RQ_2), our results suggest that classifiers trained on a dataset containing methods in one programming language (Java) have reduced performance when applied to datasets with functionally equivalent methods implemented in a different programming language (Python, C++). Hence, classification models built according to the proposed PMR approach may not be transferable across languages.

Table 23. Comparison of PMR performance (AUC and BSR) achieved by Kanewala *et al.* and when using classifiers developed by us starting from DS_{JK}

MR	**Feat**$^{\perp}$	**Performance measurements**					
		AUC			**BSR**		
		[29]	DS_{JK}	Error$^{\pm}$	[29]	DS_{JK}	Error$^{\pm}$
ADD	NF-PF	0.81	**0.837**	**-0.027**	**0.77**	0.768	0.002
	GK	**0.83**	0.769	0.061	**0.79**	0.737	0.053
	RWK	**0.92**	0.905	0.015	**0.85**	0.843	0.007
MUL	NF-PF	0.73	**0.742**	**-0.012**	0.65	0.631	0.019
	GK	**0.78**	0.775	0.005	**0.69**	0.689	0.001
	RWK	0.83	**0.846**	**-0.016**	0.74	**0.774**	**-0.034**
PER	NF-PF	0.93	**0.945**	**-0.015**	0.83	**0.847**	**-0.017**
	GK	**0.91**	0.872	0.038	0.83	**0.853**	**-0.023**
	RWK	0.95	**0.963**	**-0.013**	**0.87**	0.757	0.113
INC	NF-PF	0.84	**0.845**	**-0.005**	**0.80**	0.793	0.007
	GK	**0.88**	0.850	0.030	0.75	**0.762**	**-0.012**
	RWK	**0.89**	0.862	0.028	**0.79**	0.673	0.117
EXC	NF-PF	**0.78**	0.768	0.012	0.75	**0.868**	**-0.118**
	GK	0.78	**0.873**	**-0.093**	0.74	**0.758**	**-0.018**
	RWK	**0.90**	0.731	0.169	**0.79**	**0.790**	**0.000**
INV	NF-PF	0.84	**0.905**	**-0.065**	0.64	**0.656**	**-0.016**
	GK	**0.68**	0.671	0.009	0.66	**0.679**	**-0.019**
	RWK	0.76	**0.769**	**-0.009**	0.74	**0.787**	**-0.047**

$^{\perp}$Feature extraction approach, $^{\pm}$Error ([29]$-DS_{JK}$)

A.2.3. RQ_3 How well do classifiers predict matching methods for Python and C++ methods when developed from source code in the respective target languages?

Table 25 reports the results of using the PMR approach to develop classifiers separately for each programming language (Python and C++). Comparing Table 24 and Table 25 indicates that the performance improves remarkably when using models that are trained specifically to also consider the implementation characteristics stemming from the different programming languages.

Even though the performance has improved by developing language specific classifiers, the results for Python and C++ are generally below the results achieved for Java, with the results for C++ being consistently the worst.

> With regards to *generalisability* (RQ_3), our results suggest the PMR approach can be applied for different programming languages when the classifiers are re-trained on the specific target language. The slightly lower performance, esp. for C++, needs further exploration of the data and the choice of model parameter settings (tuning).

Table 24. Performance of SVM models when trained with DS_{JV} and tested with DS_{PY} and DS_{C++}

MR	Feat[⊥]	Performance measurements											
		Accuracy		Precision		Recall		f-measure		AUC		BSR	
		DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}
ADD	NF-PF	0.575	0.459	0.572	0.522	0.555	0.551	0.554	0.473	0.551	0.529	0.563	0.466
	GK	0.564	0.447	0.532	0.426	0.543	0.470	0.531	0.473	0.547	0.452	0.561	0.427
	RWK	0.544	0.468	0.526	0.474	0.593	0.403	0.500	0.483	0.550	0.466	0.503	0.414
MUL	NF-PF	0.494	0.588	0.627	0.596	0.495	0.639	0.522	0.658	0.623	0.574	0.652	0.648
	GK	0.460	0.472	0.463	0.436	0.477	0.392	0.479	0.400	0.480	0.431	0.475	0.478
	RWK	0.499	0.388	0.499	0.388	0.492	0.387	0.488	0.393	0.490	0.393	0.499	0.384
PER	NF-PF	0.521	0.445	0.503	0.403	0.517	0.411	0.507	0.570	0.535	0.519	0.542	0.545
	GK	0.520	0.458	0.525	0.398	0.563	0.431	0.546	0.392	0.499	0.399	0.535	0.454
	RWK	0.515	0.424	0.515	0.540	0.503	0.471	0.517	0.496	0.516	0.413	0.532	0.514
INC	NF-PF	0.579	0.578	0.576	0.527	0.580	0.496	0.580	0.590	0.597	0.603	0.577	0.477
	GK	0.578	0.518	0.588	0.501	0.597	0.576	0.579	0.476	0.582	0.558	0.594	0.587
	RWK	0.536	0.521	0.525	0.444	0.508	0.528	0.536	0.526	0.512	0.478	0.500	0.486
EXC	NF-PF	0.637	0.440	0.639	0.497	0.535	0.502	0.591	0.507	0.600	0.525	0.639	0.478
	GK	0.597	0.561	0.648	0.506	0.592	0.552	0.649	0.494	0.610	0.467	0.658	0.492
	RWK	0.568	0.548	0.579	0.564	0.613	0.610	0.579	0.627	0.570	0.590	0.637	0.562
INV	NF-PF	0.531	0.493	0.534	0.422	0.525	0.433	0.502	0.471	0.514	0.411	0.512	0.491
	GK	0.472	0.411	0.478	0.395	0.473	0.401	0.477	0.421	0.468	0.403	0.469	0.459
	RWK	0.470	0.417	0.507	0.378	0.529	0.400	0.465	0.402	0.435	0.333	0.461	0.352

[⊥]Feature extraction approach, **NF-PF:** Node Feature - Path Feature, **GK:** Graphnet Kernel, **RWK:** Random Walk Kernel

Table 25. Performance of SVM models for DS_{PY} and DS_{C++} datasets

MR	Feat[⊥]	Performance measurements											
		Accuracy		Precision		Recall		f-measure		AUC		BSR	
		DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}	DS_{PY}	DS_{C++}
ADD	NF-PF	0.706	0.577	0.742	0.660	0.748	0.590	0.683	0.645	0.723	0.691	0.760	0.653
	GK	0.724	0.599	0.671	0.708	0.713	0.655	0.757	0.606	0.730	0.652	0.798	0.667
	RWK	0.737	0.738	0.653	0.720	0.699	0.797	0.668	0.730	0.693	0.750	0.726	0.725
MUL	NF-PF	0.670	0.611	0.652	0.623	0.701	0.584	0.787	0.688	0.746	0.594	0.656	0.580
	GK	0.613	0.658	0.627	0.657	0.659	0.648	0.643	0.561	0.663	0.607	0.663	0.625
	RWK	0.727	0.795	0.732	0.742	0.640	0.685	0.726	0.715	0.686	0.735	0.721	0.677
PER	NF-PF	0.818	0.732	0.822	0.702	0.820	0.778	0.755	0.763	0.769	0.754	0.865	0.754
	GK	0.835	0.777	0.785	0.725	0.820	0.830	0.802	0.752	0.797	0.717	0.829	0.694
	RWK	0.869	0.824	0.856	0.853	0.811	0.877	0.862	0.755	0.796	0.735	0.798	0.840
INC	NF-PF	0.746	0.677	0.734	0.684	0.789	0.661	0.796	0.705	0.789	0.647	0.792	0.619
	GK	0.671	0.754	0.659	0.713	0.685	0.756	0.685	0.781	0.682	0.772	0.681	0.672
	RWK	0.785	0.760	0.793	0.721	0.808	0.784	0.746	0.682	0.804	0.753	0.793	0.694
EXC	NF-PF	0.734	0.635	0.694	0.649	0.713	0.652	0.725	0.690	0.715	0.732	0.737	0.732
	GK	0.752	0.698	0.735	0.681	0.703	0.745	0.725	0.742	0.764	0.760	0.734	0.673
	RWK	0.791	0.805	0.805	0.834	0.782	0.794	0.788	0.786	0.808	0.811	0.780	0.753
INV	NF-PF	0.606	0.571	0.671	0.543	0.668	0.539	0.661	0.594	0.673	0.559	0.672	0.538
	GK	0.582	0.595	0.598	0.620	0.589	0.586	0.558	0.579	0.604	0.633	0.599	0.624
	RWK	0.683	0.611	0.673	0.629	0.717	0.614	0.648	0.649	0.675	0.711	0.712	0.708

[⊥]Feature extraction approach, **NF-PF:** Node Feature - Path Feature, **GK:** Graphnet Kernel, **RWK:** Random Walk Kernel

A.2.4. Threats to Validity

In the context of our study, two types of threats to validity are most relevant: threats to internal and external validity.

To achieve internal validity, we used the same set of methods and of MRs as in Kanewala *et al.* [29]. For the Python and C++ datasets, we carefully checked functional equivalence of the methods with those in the original Java dataset. Given functional equivalence of the methods, we assume that the matching MRs are identical for each of the three chosen programming languages. However, this has not been verified. It is unlikely but possible that some methods have slight differences in the set of matching MRs due to the programming language. Another potential validity threat in our study is that we recreated all steps of the PMR approach using different machine learning libraries with potentially different parameter settings.

However, the performance measures in RQ_1 (Table 22) align well with the results reported in the original study. This suggests that we have understood how to correctly build the classifiers in our replication.

Regarding external validity, our study uses the same methods as in the original study but implemented in different programming languages. For the sake of generalisability, it would have been preferable to include additional methods to overcome any potential bias introduced by the selection of methods in the original study. As a consequence, our replication cannot determine the actual scope of the effectiveness of the PMR approach.

A.3. Conclusion

We closely as well as conceptually replicated the study of Kanewala *et al.* [29]. First, we reproduced the PMR approach using our own implementation of the pipeline for feature extraction and training classifiers by starting out from Java source code and creating corresponding CFGs. We showed that our classifiers perform equally well as in the original study indicating a successful replication as basis for further experiments. Second, we checked transferability of classifiers trained on methods implemented in Java to other programming languages (Python and C++). We found that the performance decreases too much to consider this approach feasible. This is caused by programming language-specific implementation details, despite relying only on features extracted from the abstract CFG representation of the methods.

Third, we demonstrated that the PMR approach can be generalised. When re-training the classifiers from scratch on Python and C++ source code, the performance we achieved was almost comparable to those from classifiers trained on Java code.

All artefacts created by us as well as all results are available in a replication package.

Appendix B. USING SOURCE CODE METRICS FOR PMR - EXTENSION OF THE PMR APPROACH

In Appendix B, we present an extension of the PMR approach by incorporating source code metrics into the classification process. The methodology, dataset, and results of this extended analysis are provided, offering insights into how source code metrics can enhance the prediction and classification of metamorphic relations across different systems. The goal is to determine whether and how it is possible to achieve similar or better performance than that obtained by Kanewala *et al.* [29], [30], but using features extracted from source code rather than those derived from the CFG.

B.1. Methodology

In this extension of the PMR approach, we focus on automated MR identification, following the PMR approach but using source code metrics instead of CFG information. The research questions are addressed through the analysis of results from multiple implementations of the PMR procedure. Therefore, in this section, we first present the research questions (Appendix B.1.1), then PMR procedure, which is slightly different from the one proposed by Kanewala *et al.* [29], [30] (Appendix B.1.2). Then, we present the labelled dataset used in our study (Appendix B.1.3). Finally, we present the performance measures used in this study (Appendix B.1.4)

B.1.1. Research Questions

In the context of our study, we answer the following research questions:

- **RQ_1:** What set of source code based features provides the best PMR performance?
- **RQ_2:** Does PMR performance improve when using source code based features instead of CFG-based features?

we focus on PMR feature engineering. In particular, we are interested in understanding whether and how it is possible to achieve a similar or better performance than that obtained by Kanewala *et al.* [29], [30], but using features extracted from source code rather than features extracted from the CFG.

B.1.2. PMR-Software-Metrics Based Procedure

Figure 31 shows the PMR-Software-Metrics procedure. The PMR Software Metrics procedure consists of three phases. *Phase I* is responsible for extracting metrics of the method source code. The output of this phase is a csv file with 21 source code metrics related features per method. *Phase II* is in charge of preparing the data according to the requirements of the ML algorithms. For example,

encoding categorical features. Also, each method is labelled with elements from the set of pre-defined MRs. *Phase III* is in charge of training and evaluating the binary classification models that predict whether a specific MR is applicable to the unit testing of a specific method. Below we describe each phase in detail.

Phase I - Feature extraction. in this phase, a list of features from various source code metrics is retrieved for each method. A source code metric is a quantitative measure of a software system's attribute. Measuring software complexity [104], assessing software maintainability [105], measuring software quality [106], and other applications rely on source code metrics. Recent studies have demonstrated the usefulness of employing source code metrics in a variety of research areas, including defect prediction [107], and time cost reduction in mutation testing [108]–[110]. In our implementation, a total of 21 source code metrics are used, as shown in Table 26. We use SCminer[1], which is an open-source tool for mining source code metrics at method level that support three different programming languages (Java, C++, and Python).

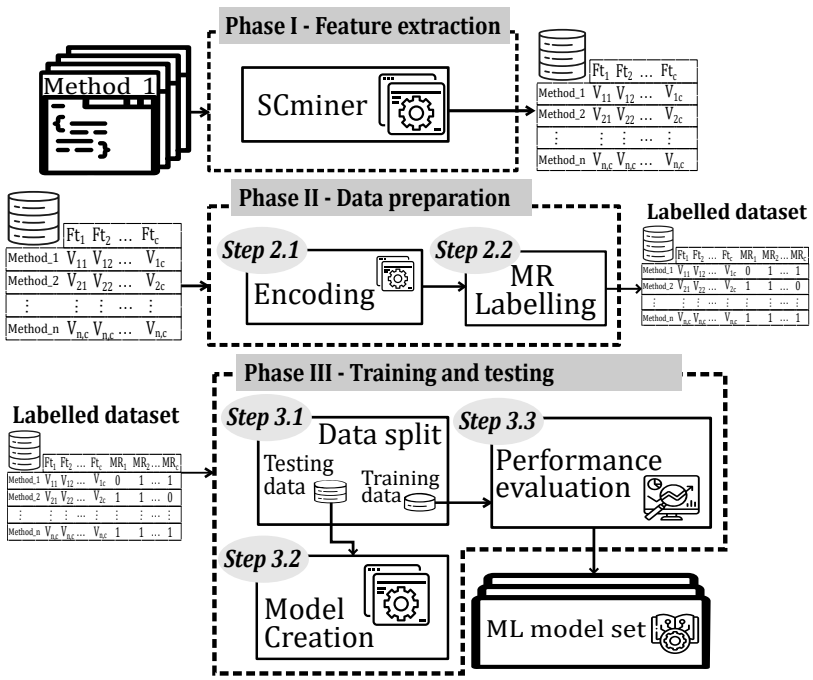

Figure 31. PMR procedure with new feature set

Phase II – Data preparation. This phase is made up of two steps:

Step 2.1 – Encoding: This step is in charge of preparing the data according to the requirements of the ML algorithms. For instance, encoding categorical features. Also, a further feature selection analysis can be done. Feature selection is an extra activity that allows exploring the different performances of ML models when different features are used.

[1]https://github.com/aduquet/SCminer

Table 26. Software metrics

Metric	Description
tloc	Total number of lines of code
sloc_whbl	Total number of code lines without blank lines
nloc	Total number of code lines without comments or libraries statements
nloc_whbl	Total number of code lines without blank lines, comments or libraries statements
sloc_statements	Total number of lines with code statements
token_count	Token is the word and operators, etc.
start_line	First code line
end_line	Last code line
full_parameters	It provides the full input parameters including the variable name
numArg	Number of inputs arguments
dataArg	Inputs data type, *e.g., int array, int, float, etc.*
numOper	Number of arithmetical operators
numOperands	Number of operands, *e.g., variables, numeric and string constants*
total_Var	Total number of variables declared
numLoops	Number of loops
CCN	Cyclomatic Complexity Number, which is the number of possible alternative paths through a piece of code
numMethCall	Number of external methods called
has_return	Tells if the method has a return value
totalReturn	Tells how many return statements the method has
returnDataType	Return data type, *e.g., int array, int, float, etc.*
ext	extention file (Java, C++, or Python)

Step 2.2 – Labelling MRs: Since PMR employs supervised learning classification techniques, which need the usage of a labelled dataset to give instances for learning. After the feature extraction phase and encoding step, the training dataset is constructed by manually labelling each method with suitable MRs. Depending on whether a certain MR does or does not satisfy the method, the method is labelled with a 1 or a 0 for such MR.

Phase III – Training and testing. This step entails extracting information from data using one or more supervised ML algorithms, or a combination of them. There are three steps that must be taken.

Step 3.1 – Data split: This step is responsible for splitting the dataset into two subsets: a training set and a test set. The training set is used to create the prediction model, while the test set is used to evaluate the performance of the created prediction model.

Step 3.2 – Model creation refers to the process of building predictive models. Selecting an appropriate modelling technique is crucial for both the training and

prediction phases in any ML application, including the PMR approach. In this study, five popular classification algorithms are evaluated for their application to PMR, including:

- Random Forest (RF)
- Support Vector Machine (SVM) with linear kernel
- Decision Trees (DT)
- Gaussian Naıve Bayes (GNB)
- Logistic Regression (LR)

Step 3.3 – Performance evaluation: This step measures the performance of the created prediction models. To assess classification performance, we utilised stratified *k*–fold crossvalidation. The *k*–fold crossvalidation approach assesses how well a prediction model performs on previously unknown data. The data set is randomly partitioned into *k* subgroups in *k*–fold cross-validation. Then, *k*–1 subsets are utilised to develop the predictive model (training), and the remaining subset is used to assess the predictive model's performance (testing). This procedure is performed *k* times, with each of the *k* subsets being used to assess performance. In stratified *k*–fold cross-validation, *k*–folds are partitioned so that they include roughly the same percentage of positive (functions that display a specific MR) and negative (functions that do not exhibit a specific metamorphic relation) samples as the original data set.

B.1.3. Dataset and pre-defined set of MRs

Kanewala *et al.* [29] provide a dataset with 100 Java methods in their CFG representation[2]. Instead of using the CFGs, we built a code corpus containing the source code of the same 100 Java methods. The methods are from the open-source libraries *Colt Project* [67], an open-source library written for high-performance scientific and technical computing, *Apache Mahou* [68], a machine learning library, *Apache Commons Mathematics* [69], a library of mathematics and statistics components, and *Java Collections* [70], a framework that provides an architecture to store and manipulate the group of objects. All of these libraries are written in Java. To obtain the source code of the methods presented in the form of CFGs by Kanewala *et al.*, we search in the different libraries for the name of the method.

We evaluate the performance of our PMR-software-metric-based implementation using prediction *recall*, *accuracy*, *precision*, *F1-score*, and *AUC-ROC* for each subject using 10-fold cross validation and then summarise (using the mean) those statistics to compare the performance of different classification algorithms. To be able to make a fair comparison, we use the same set of six pre-defined MRs as in the original study [29].

[2]http://www.cs.colostate.edu/saxs/MRpred/functions

Table 12 lists the MRs used, the changes in the inputs and the expected outputs, and the total number of methods to which a specific MR applies. Details of the set of methods, *i.e.,* method name, library to which it belongs, and the MRs that apply, have been made available in github[3].

B.1.4. Performance Measures

We evaluate the performance of our PMR implementation using prediction *recall*, *accuracy*, *precision*, *F1-score*, and *AUC-ROC* for each subject using 10-fold cross validation and then summarise (using the mean) those statistics to compare the performance of different classification algorithms. We denote a classification output in which a specific MR_n satisfies the method m as the *positive class* and a classification output in which specific MR_n does not satisfy the method m as the *negative class*. Using this notation, each standard performance measure is expressed as a function of the counts of elements in the *Confusion Matrix* defined as follows (denote TP as true positive, TN as true negative, FP as false positive, FN as false negative):

- recall = TP / (TP + FN)
- accuracy = (TP + TN) / (TP + FP + TN + FN)
- precision = TP / (TP + FP)
- F1-score is the harmonic mean of precision and recall
- AUC-ROC, area under the receiver operating characteristic (ROC) curve

B.2. Results

The full set of data generated during our experiments as well as all scripts can be found in our GitHub repo[4].

B.2.1. RQ_1: What set of source code based features provides the best PMR performance?

As a first step to answering ARQ_4, we explore whether using a subset of the maximum set of 21 features helps improve the performance of each MR classifier. To do this, an RF classifier is used to calculate the importance score of each feature. Note that, taking randomness into account, we built RF classifiers ten times and then averaged the importance score for each feature. Figure 32 shows the feature importance score per MR, Table 27 and Table 28 lists the absolute importance values per MR. In both, figure and tables, the features are ranked from highest to lowest score. From Figure 32, Table 27 and Table 28 one can see that regardless of the MR, the importance ranking of features follows a fairly uniform pattern. It seems that from the third-ranked feature, *i.e.,* "C: tloc", regardless of the MR,

[3]Link to methods description

[4]VST22_PMR-SourceCodeMetrics-E536/

Table 27. Feature importance score absolute values I

MR	A	B	C	D	E	F	G	H	I	J
ADD	0.92	1.00	0.71	0.40	0.38	0.31	0.30	0.30	0.29	0.29
EXC	1.00	0.86	0.62	0.54	0.48	0.48	0.46	0.45	0.45	0.43
INC	1.00	0.81	0.53	0.48	0.46	0.45	0.44	0.43	0.43	0.42
MUL	1.00	0.72	0.67	0.41	0.33	0.32	0.30	0.28	0.27	0.26
PER	0.57	1.00	0.54	0.41	0.38	0.36	0.24	0.23	0.23	0.22
INV	1.00	0.68	0.56	0.41	0.35	0.32	0.31	0.30	0.29	0.28
AVG	0.91	0.85	0.60	0.44	0.40	0.37	0.34	0.33	0.33	0.32

AVG: Average, **A:** dataArg, **B:** CCN, **C:** tloc, **D:** sloc-whbl, **E:** sloc-statements
F: nloc-whbl, **G:** nloc, **H:** token-count, **I:** start-line, **J:** end-line

Table 28. Feature importance score absolute values II

MR	K	L	M	N	O	P	Q	R	S	T	U
ADD	0.28	0.28	0.27	0.26	0.24	0.23	0.22	0.21	0.00	0.00	0.00
EXC	0.42	0.30	0.28	0.26	0.24	0.24	0.22	0.21	0.00	0.00	0.00
INC	0.41	0.33	0.28	0.25	0.22	0.21	0.20	0.16	0.00	0.00	0.00
MUL	0.26	0.25	0.25	0.24	0.23	0.23	0.21	0.15	0.00	0.00	0.00
PER	0.22	0.21	0.20	0.19	0.18	0.15	0.14	0.09	0.00	0.00	0.00
INV	0.27	0.26	0.25	0.22	0.21	0.20	0.19	0.18	0.00	0.00	0.00
AVG	0.31	0.27	0.25	0.24	0.22	0.21	0.20	0.17	0.00	0.00	0.00

K: numArg, **L:** numLoops,**M:** totalVar,**N:** numOper, **O:** numMethCall, **P:** hasReturn
Q: totalReturn, **R:** numOperands, **S:** returnDataType, **T:** ext, **U:** full-Parameters

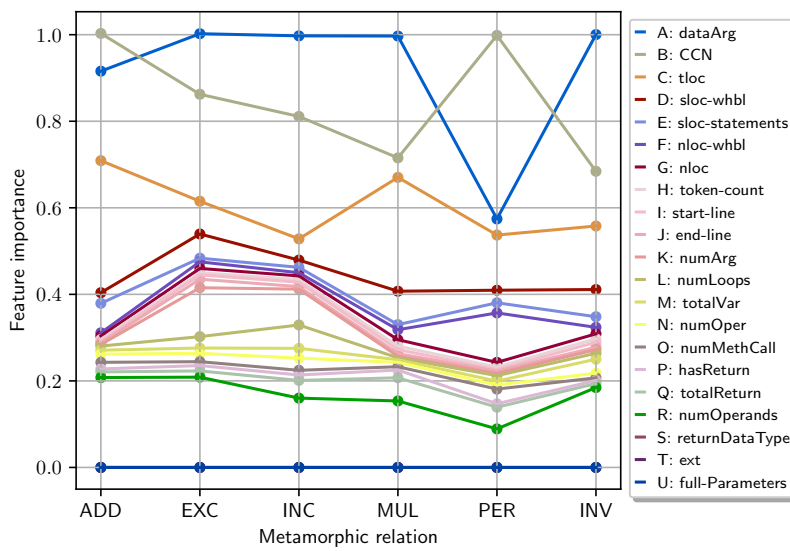

Figure 32. Feature importance score per MR

features are ranked in the same order. Moreover, for the last three features, *i.e.,* “S: returnData Type", “T: ext", and “U: full-Parameters", the score is zero for all MRs. Also, it is important to highlight that from the fourth feature for the MRs:

MUL, PER, and INV, their scores are quite close to each other for the same features. This happens also for the MRs: ADD, EXC, and INC, but from the eleventh feature, *i.e.*, "L: numLoop", onward. Regarding the top-3 features with the best scores, the MRs: EXC, INC, MUL and INV follow the same pattern concerning the order of the feature importance score, this is "A: dataArg", "B: CCN" and "C: tloc". In MRs: ADD and PER, the order of importance differ because "B: CCN" is the highest score instead of "A: dataArg". It could be said that this is the only exception to the general ranking pattern we found.

Table 29. AUC-ROC and precision absolute values per MR using the top-ranked n features in a RF classifier

Metric	MR	Top-ranked n features						
		$Feat_3$	$Feat_6$	$Feat_9$	$Feat_{12}$	$Feat_{15}$	$Feat_{18}$	$Feat_{21}$
AUC*	ADD	0.620	0.590	0.620	**0.886**	0.620	0.590	0.590
	EXC	0.629	0.593	0.583	**0.833**	0.589	0.623	0.666
	INC	**0.720**	0.675	**0.720**	0.717	0.675	0.675	0.694
	MUL	0.649	0.518	0.518	**0.762**	0.518	0.491	0.578
	PER	**0.763**	0.725	0.725	0.747	0.725	0.641	0.747
	INV	0.595	0.611	0.588	0.625	0.636	0.545	**0.640**
Prec$^{\pm}$	ADD	0.627	0.613	0.640	0.767	0.740	0.729	**0.769**
	EXC	0.667	0.651	0.693	**0.866**	0.667	0.688	0.614
	INC	0.767	0.733	0.733	**0.888**	0.727	0.761	0.652
	MUL	**0.875**	0.833	0.854	0.853	0.675	0.630	0.657
	PER	0.625	0.761	0.805	**0.814**	0.625	0.625	0.625
	INV	0.675	0.625	0.714	**0.833**	0.75	0.600	0.675

Feat: Feature, ***AUC:** AUC-ROC, $^{\pm}$**Prec:** Precision

After ranking the features from highest to lowest according to their importance score, we explored the gradual selection of best-ranked features. Also, we attempted to discover the best subset in the reduced subject set using 10-fold cross-validation and RF classifier. We started with a subset with the three best-ranked features; then we augment the subset with the following three features, and so on. In total, we evaluated seven subsets for each MR. Each time, the next three highest-ranked features were added until reaching the total set of 21 features.

Table 29 shows the performance values obtained using the RF classifier in terms of AUC-ROC and Precision when the classifier is trained with a different feature subsets. In this analysis, we want to know which subset of features is the best based on the AUC-ROC and Precision performance measures. AUC-ROC indicates the extent to which the model can distinguish between classes. The higher the AUC-ROC, the better the model will predict the positive class as positive and the negative class as negative. In our context, the positive case is the one when an MR applies to a method. Since we plan to use the PMR approach for generating initial test cases for methods that are yet lacking tests, it is more important that we avoid the occurrence false positives. A false positive would result in generating tests based on a MR that actually is not applicable.

The performance measure *Precision* is commonly used to assess the capability of a binary classifier to predict the positive case correctly. Therefore, we are particularly interested in this measure when comparing the performance of the various classifiers we build.

With regards to AUC-ROC, Table 29 shows that three out of six MRs, i.e., ADD, EXC, and MUL, have the best results with the top-ranked 12 features, with a difference greater than 0.2 to all other feature sets. For MRs INC and PER, the highest values for AUC-ROC are achieved when the 3 top-ranked features are used. However, the difference to the performance using 12 top-ranked features is less than about 0.01. For MR INV, the best AUC-ROC score is when the complete set of features is used. However, as for INC and PER, the difference with the 12 top-ranked features is rather small (0.015). Therefore, based on the AUC-ROC measure, it seems to be reasonable to simply use the 12 top-ranked features across the board for all MRs. With regards to *Precision*, Table 29 shows that for four out of six MRs, i.e., EXC, INC, PER and INV, the best performance is achieved with the 12 top-ranked features. The highest precision value for the MR ADD is obtained when all 21 features are used. However, the difference between 21 features and 12 top-ranked features is only 0.002, approximately. This indicates that performance-based on precision does not vary significantly if the top 12 features are used.

In the next step, we explore the PMR approach using four different classification models in addition to RF, i.e., DT, GNB, SVM and LG. Each classifier is trained with the top-ranked 3 and 12 features, and the full set of 21 features. We include the sets of 3 and 21 features in the analysis to double-check whether the choice of 12-features is also the best for other classifiers than RF. We evaluate the performance of each classifier using 10-fold cross-validation. Specifically, 70% of the instances' datasets are used for training in each fold, and the remaining 30% are used for testing. The prediction statistics (*accuracy, precision, recall, F1 score, and AUC-ROC*) are recorded for each fold. The average values of these statistics for each classifier and each MR are listed in Table 30.

As expected, Table 30 confirms that using the 12 top-ranked features is almost always the best choice across the board (best performance is printed in bold). In those cases where 12 features don't yield the best performance, the difference to the best performing feature set is always less than 0.02.

To decide which classifier is the best for each MR, we relied on the values of AUC-ROC and *Precision* when using classifiers based on 12 features.

For MR ADD, the highest average of AUC-ROC and *Precision* is achieved when using GNB (average of 0.875 and 0.914).

For MR EXC, the highest average of AUC-ROC and *Precision* is achieved when using RF (average of 0.833 and 0.866).

For MR INC, the highest average of AUC-ROC and *Precision* is achieved when using RF (average of 0.717 and 0.888).

For MR MUL, the highest average of AUC-ROC and *Precision* is achieved

Table 30. PMR performance metrics when using 3, 12, and 21 top-ranked features on RF, DT, GNB, SVM, and LG classifiers

MR	Clas*	Accuracy			Precision			Recall			F1 score			AUC-ROC		
		$Feat_3$	$Feat_{12}$	$Feat_{21}$	$Feat_3$	$Feat_{12}$	$Feat_{21}$	$Feat_3$	$Feat_{12}$	$Feat_{21}$	$Feat_3$	$Feat_{12}$	$Feat_{21}$	$Feat_3$	$Feat_{12}$	$Feat_{21}$
ADD	RF	0.764	**0.886**	0.884	0.627	0.767	**0.769**	0.972	0.971	**0.973**	0.827	**0.830**	0.824	0.620	**0.886**	0.590
	DT	0.787	**0.800**	0.680	0.769	**0.812**	0.725	0.750	**0.833**	0.666	0.692	**0.727**	0.656	0.757	**0.816**	0.698
	GNB	0.809	**0.833**	0.726	0.805	**0.914**	0.695	0.780	**0.833**	0.726	0.796	**0.823**	0.769	0.771	**0.875**	0.667
	SVM	0.740	**0.803**	0.620	0.754	**0.833**	0.675	0.729	**0.838**	0.620	0.747	**0.835**	0.659	0.753	**0.833**	0.673
	LG	0.680	**0.800**	0.680	0.750	**0.807**	0.693	0.762	**0.833**	0.690	0.762	**0.833**	0.690	0.738	**0.800**	0.675
EXC	RF	0.725	**0.800**	0.650	0.667	**0.866**	0.614	0.833	**1.000**	0.666	0.733	**0.800**	0.666	0.629	**0.833**	0.666
	DT	0.660	**0.750**	0.570	0.770	**0.844**	0.695	0.600	**0.667**	0.532	0.551	**0.602**	0.500	0.602	**0.667**	0.536
	GNB	0.690	**0.712**	0.667	0.733	**0.800**	0.666	0.600	**0.667**	0.534	0.564	**0.667**	0.461	0.595	**0.619**	0.571
	SVM	0.672	**0.704**	0.640	0.700	**0.733**	0.667	0.599	**0.667**	0.530	0.710	**0.857**	0.562	0.634	**0.694**	0.573
	LG	0.662	0.657	**0.667**	0.735	**0.844**	0.625	0.628	**0.665**	0.590	0.621	**0.671**	0.571	0.610	**0.696**	0.523
INC	RF	0.765	**0.900**	0.630	0.767	**0.888**	0.652	0.778	**0.890**	0.666	0.808	**0.888**	0.727	**0.720**	0.717	0.694
	DT	0.681	**0.750**	0.612	0.566	**0.616**	0.516	0.465	**0.596**	0.333	0.513	**0.589**	0.438	0.564	**0.605**	0.523
	GNB	0.687	**0.705**	0.668	0.733	**0.862**	0.604	0.627	**0.667**	0.587	0.604	**0.667**	0.542	0.596	**0.681**	0.510
	SVM	0.706	**0.802**	0.610	0.737	**0.806**	0.667	0.590	**0.645**	0.534	0.558	**0.667**	0.448	0.655	**0.761**	0.548
	LG	0.657	**0.701**	0.613	0.648	**0.695**	0.601	0.633	**0.668**	0.597	0.452	**0.571**	0.333	0.560	**0.690**	0.429
MUL	RF	0.725	**0.800**	0.650	**0.875**	0.853	0.657	0.912	**1.000**	0.823	0.812	**0.875**	0.748	0.649	**0.762**	0.578
	DT	0.662	**0.703**	0.620	0.749	**0.802**	0.695	0.828	**0.833**	0.822	0.776	**0.833**	0.719	0.721	**0.791**	0.651
	GNB	0.680	**0.700**	0.660	0.755	**0.834**	0.675	0.899	**0.940**	0.857	0.803	**0.823**	0.782	0.575	**0.625**	0.524
	SVM	0.707	**0.804**	0.610	0.792	**0.850**	0.733	0.845	**0.857**	0.833	0.742	**0.857**	0.626	0.726	**0.761**	0.690
	LG	0.741	**0.801**	0.680	0.772	**0.875**	0.669	0.840	**0.857**	0.823	0.817	**0.875**	0.759	0.792	**0.833**	0.750
PER	RF	0.715	**0.810**	0.620	0.625	**0.814**	0.675	0.639	**0.712**	0.566	0.726	**0.789**	0.662	**0.763**	0.725	0.747
	DT	0.708	**0.750**	0.666	0.846	**0.914**	0.777	0.721	**0.775**	0.666	0.717	**0.857**	0.576	0.809	**0.857**	0.761
	GNB	0.623	**0.700**	0.545	0.725	**0.828**	0.622	0.500	**0.666**	0.333	0.589	**0.727**	0.450	0.604	**0.642**	0.566
	SVM	0.875	**0.910**	0.840	0.837	**0.875**	0.799	0.709	**0.750**	0.667	0.698	**0.729**	0.667	0.793	**0.825**	0.761
	LG	0.765	**0.830**	0.700	0.803	**0.822**	0.783	0.688	**0.709**	0.667	0.720	**0.750**	0.690	0.745	**0.795**	0.694
INV	RF	0.655	**0.702**	0.608	0.675	**0.833**	0.675	0.788	**0.857**	0.719	0.776	**0.800**	0.751	0.595	0.625	**0.640**
	DT	0.762	**0.800**	0.600	0.703	**0.844**	0.563	0.762	**0.857**	0.667	0.691	**0.714**	0.667	0.604	**0.667**	0.541
	GNB	0.661	**0.701**	0.620	0.659	**0.833**	0.484	0.759	**0.857**	0.660	0.787	**0.823**	0.750	0.568	**0.625**	0.511
	SVM	0.776	**0.802**	0.750	0.768	**0.844**	0.692	0.799	**0.833**	0.764	0.813	**0.857**	0.769	0.762	**0.857**	0.667
	LG	0.768	**0.860**	0.676	0.728	**0.761**	0.695	0.794	**0.857**	0.731	0.767	**0.857**	0.676	0.714	**0.762**	0.667

*Classifier

when using LG (average of 0.833 and 0.875).

For MR PER, the highest average of AUC-ROC and *Precision* is achieved when using DT (average of 0.857 and 0.914).

For MR INV, the highest average of AUC-ROC and *Precision* is achieved when using SVM with linear kernel (average of 0.857 and 0.844).

In summary, the results show that there is not one best classifier. RF is best for two MRs and each of the other classifiers is best for exactly one MR.

With regards to RQ_1, our results indicate that we achieve almost always a performance greater than 0.8 in terms of AUC-ROC and *Precision* when predicting MRs using source code based features (the one exception is the case of AUC-ROC for MR INC). The best results are obtained with only 12 features out of 21. However, there is not one single best classifier for all MRs. With regards to *Precision*, each classifier is best for one MR at least once. With regards to AUC-ROC, with the exception of DT, each classifier is best for one MR at least once.

B.2.2. RQ_2: Does PMR performance improve when using source code based features instead of CFG-based features?

The left-hand side of Table 31 shows the AUC-ROC values obtained when SVM models are trained with three feature extraction approaches, i.e., node and path features (NF-PF), graphlet kernel (GK), and random walk kernel (RWK), as reported by Kanewala *et al.* [29]. The features used in these classifiers are CFG-related. Since precision has not been reported by Kanewala *et al.*, we must base our comparison exclusively on the AUC-ROC measure. Also, Kanewala *et al.* do not report results from other models but SVM. The right-hand side of Table 31 shows the AUC-ROC values for the five classifiers using 12 top-ranked source code based features. The highest AUC-ROC values for each MR are printed in bold font.

Table 31. Comparison between AUC-ROC values per MR obtained by Kanewala *et al.* [29] using SVM with CFG-related features (NF-PF, GK and RWK), and AUC-ROC values obtained when using RF, DT, GNB and LG, with the 12 top-ranked source code based features.

MR	Kanewala *et al.* [29]			12 top-ranked source code based feat.				
	NF-PF	GK	RWK					
	SVM	SVM	SVM	SVM	RF	DT	GNB	LG
ADD	0.81	0.83	**0.92**	0.83	0.89	0.82	0.88	0.80
EXC	0.78	0.78	**0.90**	0.69	0.83	0.67	0.62	0.70
INC	0.84	0.88	**0.89**	0.76	0.72	0.61	0.68	0.69
MUL	0.73	0.78	**0.83**	0.76	0.76	0.79	0.63	**0.83**
PER	0.93	0.91	**0.95**	0.83	0.76	0.86	0.64	0.80
INV	0.84	0.68	0.76	**0.86**	0.64	0.67	0.63	0.76

Feat: Feature, ***AUC:** AUC-ROC, $^{\pm}$**Prec:** Precision

The comparison between Kanewala *et al.* and our models shows that in five out of six cases Kanewala *et al.*'s RWK-SVM model is performing best. Only for MR INV our SVM model using 12 top-ranked source code based features and linear kernel is performing best. For MR MUL, our LG classifier achieves a tie. However, considering that the extraction of features from CFGs is more expensive than building classifiers directly from the source code (i.e., without first having to construct CFGs from the source code and then analyse them), our classifiers might still be acceptable if their performance is not much lower than that of Kanewala *et al.*'s best classifier. From Figure 33 (which plots the values presented in Table 31), one can see that not only for MR INV one of our classifiers performs best but also for all other MRs one of our classifiers has a performance close to that of Kanewala *et al.*'s best classifier.

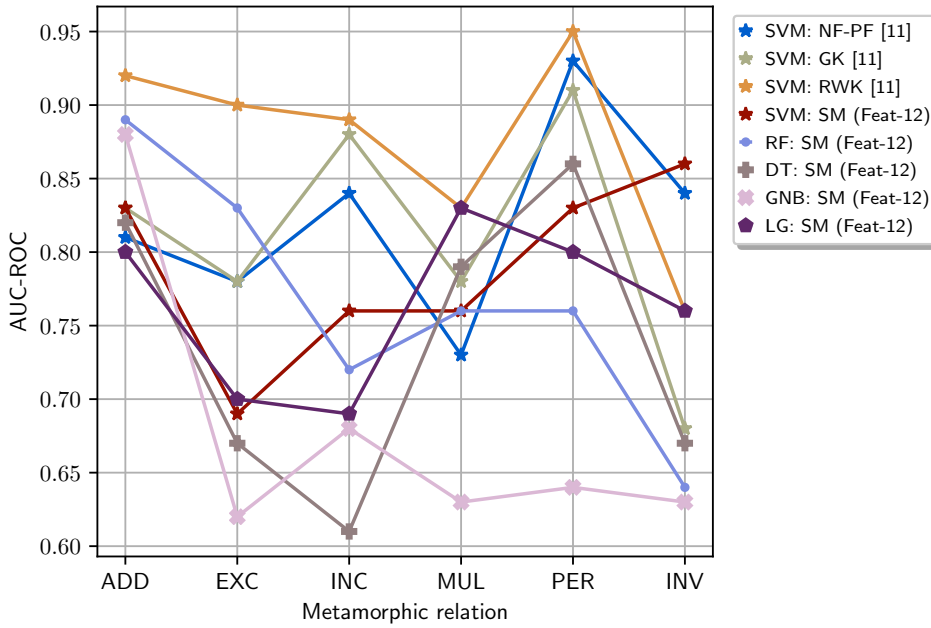

Figure 33. Comparison of the AUC-ROC results obtained by Kanewala *et al.* [29] using node- path-based features (NF-PF), Graphlet Kernel (GK) and Random Walk Kernel (RWK) in SVMs, and the AUC-ROC results obtained in this work using the source code Metrics (SM) in SVM, RF, DT, GNB, and LG

Regarding RQ_2, our results indicate that classifiers using source code based features most of the time cannot achieve better performance in terms of AUC-ROC. SVM classifiers using CFG-based features and the RWK are best for five out of six MRs. Only for INV, the SVM classifier using twelve source code based features (and the default linear kernel) is better than the SVM classifier with RWK proposed by Kanewala *et al.* [29]. However, the performance of classifiers using source code based features is not dramatically worse than that of the best classifier using CFG-based features. Since source code based features are much cheaper to extract, this might outweigh the small loss of performance.

B.2.3. Threats to Validity

We now discuss the four most relevant threats to validity of our study.

Internal validity. Since Kanewala *et al.*'s original study only reported the performance of their classifiers only in terms of AUC-ROC, we had to base our comparison on that measure although, for our intended application of the PMR approach, precision would be the more appropriate measure. In addition, more studies are needed to investigate how it affects the overall performance of PMR

when the same method is implemented differently, as this would directly affect feature extraction in both the original study and our modelling approach.

External validity. For the sake of fair comparison, our study uses the same set of methods as the original study but uses the source code of the method instead of its CFG representation. However, it would have been preferable to use a corpus of source code consisting of a greater number of methods. Consequently, both our study and the original one cannot determine the true extent of the efficacy of the PMR approach.

Construct validity. We used the SCmine framework to extract the source code metrics (features) at method level. SCminer is an open-source tool that uses third-party libraries. Usage of these third-party libraries represents potential threats to construct validity. To avoid this, we verified that the results produced by SCminer are correct by manually inspecting randomly selected outputs produced by the tool.

Conclusion validity. We used AUC-ROC and presicion value for evaluating the performance of the classifiers. We considered AUC-ROC and Presicion $>$ 0.80 as a good classifier. This is consistent with most of the ML literature.

B.3. Conclusion

We evaluate the performance of PMR using features related to the source code. We start by extracting 21 metrics related to the source code as features. Next, we perform features importance analysis using RF classifiers. After selecting the best set of features, we evaluate them in five different classifiers to find out which is the best in terms of AUC-ROC and Precision. Finally, to see if source code-related features improve PMR performance when using CFG-related features; we compared the AUC-ROC results obtained by Kanewala *et al.* [29] with our own results. In summary, a total of 21 characteristics and 5 classification algorithms are evaluated in this study. All classifiers are carefully evaluated using 10-time cross-validation; To evaluate the performance of our PMT implementation, 5 performance metrics are recorded (per fold) and then averaged. Our results show that PMR can achieve results greater than 0.8 in terms of precision when predicting MRs using features based on the source code. The best results are obtained with only 12 features out of 21. However, there is no single best classifier for all MRs. Also, classifiers that use source-based features most of the time cannot perform better in terms of AUC-ROC. For this particular performance metric, the use of CFG-related functions in particular RWK is better than ours four out of six times and ties once.

ACKNOWLEDGEMENTS

First and foremost, I thank God for giving me the strength, health, opportunity, and life to pursue this research. I am extremely grateful to my family—my mom, Constanza, my dad, Arles, and my sister, Nath—for their constant support, prayers, conversations, laughter, and, above all, their love throughout my life. My success rests on the shoulders of my parents; without them, I would be nothing. Their unwavering encouragement has been the foundation of everything I have ever accomplished.

I would also like to express my deepest gratitude to my supervisor, Professor Dietmar Pfahl, for his invaluable guidance, support, and encouragement throughout this journey. Additionally, I would like to thank my office mates and friends, Hina, Kristiina and Fauzia, for the great discussions and laughter—apologies for being noisy at times! I am deeply grateful to Rudolf Ramler for welcoming me to the Software Competence Center Hagenberg (SCCH) and for introducing me to the AQUA team. The collaborations with SCCH were a key part of my PhD journey, and the experience was invaluable. Thank you, Claus Klammer, Stefan Fisher, and Andrea Walchshofer, for welcoming me to the team.

To my beloved friends Ezequiel, Avi, Orlenys, David, Iry, Manuel, Heidi, Anna-Christina, Stefan, Andrea, and Sabsi, who I had the pleasure of knowing during this journey—thank you for your support, jokes, deep discussions, trips, reels, laughter, and most importantly, lunches. Thanks for being my chosen family. Without all of you, this journey would not have been as incredible and awesome. Finally, but certainly not least, I want to quote the legendary Calvin Cordozar Broadus Jr (Snoop Dogg): *"Last but not least, I wanna thank me. I wanna thank me for believing in me. I wanna thank me for doing all this hard work. I wanna thank me for having no days off. I wanna thank me for never quitting. I wanna thank me for just being me at all times."* Thank you to everyone who has been a part of this journey.

The research reported in this thesis has been partly funded by BMK, BMAW, and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE [(FFG grant no. 892418)] part of the FFG COMET Competence Centers for Excellent Technologies Programme, as well as by the European Regional Development Fund, and grant PRG1226 of the Estonian Research Council.

SISUKOKKUVÕTE

Metamorfsete seoste klassifitseerimine, täiustamine ja järjestamine

Tarkvara testimine on oluline osa tarkvaraarenduse protsessist, mille eesmärgiks on tagada, et lõplik tarkvaratoode töötab ootuspäraselt. Tarkvara testimine on aga sageli aeganõudev, ressursimahukas ja keeruline, eriti suurte süsteemide puhul. Vaatamata edusammudele testimise automatiseerimises jäävad selle käigus püsima kaks suurt väljakutset: tõhusate testandmete genereerimine ja testandmetele vastavate väljundite korrektne määramine. Viimane on tuntud kui testioraakli probleem. Testioraakli probleem tekib, kui testitaval tarkvaral puudub oraakel. Testioraakli probleemi lahendamist on küll palju uuritud ning kuigi mudelipõhise testimise abil on saavutatud mõningaid edusamme, on probleem veel suures osas lahenduseta.

Metamorfne testimine on tarkvara testimise meetod, mille eesmärgiks on testioraakli probleemi leevendamine. Erinevalt traditsioonilistest testimistehnikatest, mis keskenduvad vaid üksikute sisend-väljund kombinatsioonide kontrollimisele tarkvaras, analüüsib metamorfne testimine seoseid sisend-väljund paaride vahel tarkvara järjestikustel käivitamistel. Selliseid seoseid sisendite ja väljundite vahel nimetatakse metamorfseteks seosteks. Metamorfsed seosed määravad, kuidas tarkvara väljundid peaksid muutuma vastavalt muutustele mis tehakse tarkvarale antud konkreetsetes sisendites. Kui vähemalt ühe kehtiva testisisendi puhul metamorfset seost rikutakse, viitab see suurele tõenäosusele, et testitav tarkvara sisaldab viga. Siiski ei tähenda seoste rikkumiste puudumine, et testitav tarkvara on veatu. Metamorfse testimise tõhusus sõltub suuresti selleks kasutatavate metamorfsete seoste kvaliteedist. Hea metamorfne seos ei määra õigesti vaid sisendi ja väljundi seoseid kehtivas sisendite ruumis, vaid peab suutma tuvastama ka tarkvara koodis tehtud vigadest tulenevaid valesid programmikäitumisi.

Efektiivsete metamorfsete seoste genereerimine ei ole lihtne ning nende loomise viisid jagunevad kahte kategooriasse: semantiliselt ja süntaktiliselt korrektsed. Semantiliselt korrektsed metamorfsed seosed nõuavad põhjalikku arusaamist testitava tarkvara käitumisest, mis saadakse enamasti tarkvara dokumentatsiooni ja ekspertteadmiste põhjal. Viimased luuakse harilikult käsitsi, mis on ajakulukas ja nõuab valdkonna kohta põhjalikke teadmisi. Süntaktiliselt korrektsed metamorfsed seosed keskenduvad aga sellele, et seos vastaks testitava tarkvara sisend-väljund struktuurile. Need seosed luuakse sageli automatiseeritud meetoditega, kasutades selleks etteantud mustreid või üldiseid metamorfseid seoseid, mis on mitmetes süsteemides rakendatavad. Kuid isegi nende lähenemisviisidega on tõhusate metamorfsete seoste valimine ja klassifitseerimine endiselt keeruline. Metamorfsete seoste klassifitseerimise automatiseerimiseks on välja pakutud mitmeid meetodeid nagu näiteks ennustatavate metamorfsete seoste (En. Predicting Metamorphic Relations, PMR) meetod. PMR klassifitseerib metamorfseteid seo-

seid masinõppe abil koodistruktuuri põhjal, kasutades programmi juhtvoograafi või selle lähtekoodi.

Kuigi eelmainitud meetodid on andnud paljulubavaid tulemusi, seisavad need silmitsi mitmete piirangutega. Esiteks tuginevad need meetodid binaarsetele klassifikaatoritele, mis vajavad õppimiseks eelmärgendatud andmekogumeid. Märgendatud andmekogumid ei pruugi aga alati kättesaadavad olla ning nende hankimine võib osutuda aeganõudvaks. Teiseks põhineb mudeli treenimiseks vajalike tunnuste eraldamise protsess juhtvoograafi või lähtekoodi mõõdikutel, mis ei pruugi arvestada koodi refaktoreerimistega. See piirang võib mõjutada PMR meetodi täpsust, kuna refaktoreerimine võib muuta koodi struktuuri ja seeläbi ka metamorfsete seoste rakendatavust. Lõpuks ei pruugi PMR-i binaarne väljund arvestada testandmetega ja nende mõjuga metamorfsete seoste rakendatavusele. See piirang tähendab, et PMR ei arvesta võimalusega, et metamorfne seos võib kehtida ainult teatud TD-de puhul, millel on spetsiifilised omadused. See võib omakorda põhjustada valepositiivseid või valenegatiivseid tulemusi seoste valikul. Nende väljakutsete tõttu on selle lõputöö eesmärk tutvustada uusi meetodeid metamorfsete seoste klassifitseerimiseks, täiustamiseks ja hindamiseks, et parandada metamorfse testimise tõhusust ja tulemuslikkust.

Lõputöö panused jagunevad kolmeks. Esimene panus, MetaTrimmer, tutvustab uut testandmetel põhinevat meetodit metamorfsete seoste klassifitseerimiseks. Erinevalt traditsioonilistest koodistruktuuril põhinevatest klassifitseerimismeetoditest hindab MetaTrimmer dünaamiliselt metamorfsete seoste käitumist programmi erinevate sisendite korral. Jättes kõrvale eelduse, et metamorfne seos kehtib universaalselt, tuvastab MetaTrimmer konkreetsed testandmete alamhulgad, mille puhul metamorfne seos kehtib ning võimaldab seega täpsemat klassifitseerimist. Teine panus, MetaTrimmer+, laiendab metamorfsete seoste analüüsi ning tuvastab testandmetes mustreid eesmärgiga täiustada segajuhtumites kasutatavaid metamorfseid seoseid .Segajuhtumites võivad metamorfsed seosed sõltuvalt sisendist näidata nii rikkumisi kui ka mitterikkumisi

MetaTrimmer+ kasutab kombinatsiooni käsitsi ülevaatusest ja assotsiatsioonireeglite kaevandamisesest, et tuletada sisendite ruumist mustreid ja selgitada, millal käituvad seosed kas alati rikkumatult või alati rikutuna. See suurendab metamorfsete seoste kasutatavust segajuhtumite puhul, võimaldades neid rakendada nii positiivsete kui ka negatiivsete testjuhtumitena, mis omakorda laiendab testikomplekti ja parandab selle tõhusust. Kolmas panus keskendub metamorfsete seoste tugevuse hindamisele nende veatuvastusvõime alusel. Integreerides MetaTrimmeri mutatsioonitestimisega, tutvustab lõputöö meetodit metamorfsete seoste järjestamiseks nende vigade leidmise tõhususe põhjal. See võimaldab testijatel prioriseerida kõige tõhusamad seosed, mis aitab testkomplekti optimeerida nii, et testikomplekti suurus vähenebilma veatuvastusvõimet ohverdamata. Meetodeid rakendati edukalt tööstuslikus juhtumiuuringus, mis tõestas nende praktilist väärtust, kuigi mutatsioonitestimise arvutuslik maksumus ja sõltuvus mitmekesistest testsisenditest jäid siiski väljakutseteks.

CURRICULUM VITAE

Personal data

Name:	Alejandra Duque Torres
Date of Birth:	02.11.1993
Nationality:	Colombian
Language skills:	Spanish (Native), English

Education

2020-2025	University of Tartu, Tartu, Estonia, Computer Science, PhD.
2017-2019	Cauca University, Popayan, Cauca, Colombia, Telematics Engineering, MSc.
2011-2017	Quindio University, Armenia, Quindio, Colombia, Electronic Engineering, BSc.

Employment

09/2024	Research and Software Engineer, Software Competence Center Hagenberg (SCCH), Hagenberg, Austria.
02/2020 - 08/2024	Junior Research Fellow, University of Tartu, Tartu, Estonia.
06/2019 - 10/2019	Data Center Administrator, Popayan, Cauca, Colombia, Cauca University.
06/2018 - 04/2019	Research Intern, School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand.
01/2016 - 07/2016	Research Intern, Centro de Investigaciones en Optica, CIO, Leon de Guanajuato, Mexico.

Scientific work

Main fields of interest:

- Software Quality
- Software Testing
- Machine Learning

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Alejandra Duque Torres
Sünniaeg:	02.11.1993
Kodakondsus:	Kolumbia
Keeleoskus:	Hispaania keel (emakeel), inglise keel

Haridus

2020 - 2025	Tartu Ülikool, Tartu, Eesti, Kolumbia, Informaatika, PhD.
2017 - 2019	Cauca Ülikool, Popayan, Cauca, Telemaatika insener, MSc.
2011 - 2017	Quindio Ülikool, Armenia, Quindio, Kolumbia, Elektrotehnika, BSc.

Teenistuskäik

09/2024	Teadur ja tarkvarainsener, Hagenbergi tarkvara pädevuse keskus, Hagenberg im Mühlkreis, Austria.
02/2020 - 08/2024	Nooremteadur, Tartu Ülikool, Tartu, Eesti
06/2019 - 10/2019	Andmekeskuse administraator, Popayan, Kolumbia Cauca Cauca Ülikool.
06/2018 - 04/2019	Külalisteadur, Inseneri ja arvutiteaduse kool, Victoria Ülikool, Wellington, Uus-Meremaa.
01/2016 - 07/2016	Külalisteadur, Optika uurimiskeskus, CIO, Leon de Guanajuato, Mehhiko.

Teadustegevus

Peamised uurimisvaldkonnad:

- Tarkvara kvaliteet
- Tarkvara testimine
- Masinõpe

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.
41. **Mohammad Anagreh.** Privacy-Preserving Parallel Computations for Graph Problems. Tartu 2023, 181 p.
42. **Rahul Goel.** Mining Social Well-being Using Mobile Data. Tartu 2023, 104 p.

43. **Anti Ingel.** Algorithms using information theory: classification in brain-computer interfaces and characterising reinforcement-learning agents. Tartu 2023, 142 p.
44. **Shakshi Sharma.** Fighting Misinformation in the Digital Age: A Comprehensive Strategy for Characterizing, Identifying, and Mitigating Misinformation on Online Social Media Platforms. Tartu 2023, 158 p.
45. **Kristiina Rahkema.** Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects. Tartu 2023, 182 p.
46. **Ivan Slobozhan.** Studying Online Social Media Engagement in CIS Countries during Protests, Mass Demonstrations and War. Tartu 2023, 81 p.
47. **Nurlan Kerimov.** Building a catalogue of molecular quantitative trait loci to interpret complex trait associations. Tartu 2023, 248 p.
48. **Pavlo Tertychnyi.** Machine Learning Methods for Anti-Money Laundering Monitoring. Tartu 2023, 117 p.
49. **Abasi-amefon Obot Affia.** A Framework and Teaching Approach for IoT Security Risk Management. Tartu 2023, 180 p.
50. **Raimond-Hendrik Tunnel.** Video Game Design and Development Bachelor's Curriculum for Estonia. Tartu 2024, 137 p.
51. **Ahto Salumets.** Bioinformatics analysis of various aspects in immunology. Tartu 2024, 198 p.
52. **Mohammed Abdulhameed Shaif Ali.** Deep Learning Methods for Cell Microscopy Image Analysis. Tartu 2024, 143 p.
53. **Pille Pullonen-Raudvere**. Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation. Tartu 2024, 265 p.
54. **Marili Rõõm.** Multiple approaches to learners' success and factors affecting it in computer programming MOOCs. Tartu 2024, 170 p.
55. **Shivananda Rangappa Poojara.** Design and Orchestration of Scalable, Event-Driven Serverless Data Pipelines for Internet of Things (IoT) Applications. Tartu 2024, 172 p.
56. **Hassan Abdulgaleel Hassan Salim Eldeeb.** Empowering Machine Learning Pipelines with Automated Feature Engineering. Tartu 2024, 121 p.
57. **Muhammad Uzair.** Soft decision making for agri-food 4.0. Tartu 2024, 158 p.
58. **Kirill Milintsevich.** Estimation of Depression Level from Text: Symptom-Based Approach, External Knowledge, Dataset Validity. Tartu 2024, 130 p.
59. **Maksym Del.** Multilingual and Multi-Domain Representational Patterns Across Trpansformer-Based Models. Tartu 2024, 131 p.
60. **Kristo Raun.** Adaptive Out-of-order Handling in Streaming Conformance Checking. Tartu 2024, 118 p.
61. **Toivo Vajakas.** Towards integration of mobile network data into analyzing human mobility. Tartu 2024, 103 p.
62. **Katsiaryna Lashkevich.** Data-Driven Analysis and Optimization of Waiting Times in Business Processes. Tartu 2024, 169 p.