Faculty of Mathematics and Informatics

VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

SOFTWARE ENGINEERING

Metamorphic Relation Based Test Case Generation Method

Metamorfiniais ryšiais grįstas testų generavimo metodas

Master Thesis

Author:	Lukaš Michnevič
Supervisor:	asist. dr. Tomas Plankis
Reviewer:	asist. dr. Vytautas Valaitis

Vilnius – 2025

Contents

Santrauka

Metamorfinio testavimo metodas leidžia testuoti programinį kodą negeneruojant testų orakulų, t.y. tikėtinų testų rezultatų, su kuriais testo metu lyginami programos grąžinti rezultatai. Praktikoje, metamorfiniai ryšiai – tai silpna programos specifikacija, kuri koduoja tam tikrus veiksmus, kurie turi įvykti, jei tenkinamos būtinos sąlygos. Dažniausiai, MR ieškomi patyrusių testavimo inžinierių, kurie išmano dalykinę sritį bei turi gilesnių žinių apie testuojamą projektą. Automatizuotos metamorfinių ryšių (MR) paieškos procedūros yra sunkiai universalizuojamos dėl įvairių programos įgyvendinimo būdų.

Šiame straipsnyje siūloma Aya – C++ biblioteka, galinti aptikti MR bet kurioje funkcijoje, tikrindama ją pagal vartotojo pateiktus įvesties duomenis ir įvesties transformacijas. Įrodyta, kad Aya sugeba aptikti tuos pačius metamorfinius ryšius, kurie buvo rasti analogiškuose paieškos metodais pagrįstuose tyrimuose, todėl ji tinkama tolesniems eksperimentams ir tobulinimams, pavyzdžiui, testuojant apgręžiamojo skaičiavimo mechanizmus, tokius kaip kodavimo/dekodavimo ciklai ir kai kurie matricų skaičiavimai. Be to, Aya gali aptikti metamorfinius ryšius programos vykdymo metu, užfiksuodama tiesioginius kiekvieno sukurto MR rezultatus, efektyviai veikdama kaip savarankiškas testas. Norint universaliai užfiksuoti MR bet kurioms funkcijoms, buvo panaudotos metaprogramavimo technikos, tokios kaip C++ šablonai, leidžiantys išlaikyti MR generavimo paprastumą vartotojui, tuo pačiu suteikiant galimybę generuoti metamorfinius ryšius žymiai platesniam naudojimo atvejų spektrui.

Įrodyta, kad ilgos transformacijų grandinės, taikomos pradiniams įvesties ir išvesties duomenims, nėra naudingesnės už trumpesnes grandines – testuotoje programinėje įrangoje vidutinis mutacijos balas tiek trumpoms (1–2 transformacijos grandinėje), tiek ilgoms (3 ir daugiau) buvo apytiksliai vienodas: 50-70% sudėtingesnėms funkcijoms, ir beveik 100 % paprastesnėms matematinėms funkcijoms.

Raktiniai žodžiai Testų generavimas, Metamorfinis ryšys, Automatizuotas testavimas

Abstract

Metamorphic testing allows testing software that is typically hard to test extensively since it does not rely on test oracles to verify the correctness of the Software Under Test. In practice, metamorphic relations—the weak specification of a program that encodes certain actions that must occur if required conditions are met — are detected mostly manually by experienced software testing practitioners. Automated metamorphic relation (MR) search procedures are hard to universalize due to the various ways a program can be implemented.

This paper proposes *Aya* – a C++ library capable of detecting MRs in an arbitrary function by checking it against user-provided inputs and input transformations. It was shown that *Aya* can detect the same metamorphic relations as they were discovered in analogous Search-Based methods, positioning it for further experimentation and improvements, such as testing reversible computation mechanisms like encoding/decoding cycles and matrix calculations. Moreover, *Aya* can detect metamorphic relations during program runtime, capturing immediate results of each MR produced, effectively functioning as a standalone test. To help universally capture MRs for arbitrary functions, meta-programming techniques like C++ templates were deployed, allowing the user to retain the simplicity of MR generation while providing an option to generate metamorphic relations for a significantly larger number of use-cases.

It was shown that large chains of transformers applied to initial inputs and outputs are not more helpful than shorter chains – in tested software, average mutation score for both short (1-2 transforms in a chain) to long (3+) was roughly the same: 50-70% for more complex functions, and nearly 100% for simpler mathematical functions.

Keywords Test Case Generation, Metamorphic Relation, Automated Testing

Introduction

The most common approach in automated testing is running suites of unit, integration, system, and other tests against the selected revision in a codebase. It could be a continuous integration server running these suites or a developer running those tests on their device. However, with the growing complexity of software, typical unit test coverage may not be enough, especially when the same tests pass consistently on different change sets. Although existing test coverage might constantly pass with new code added, it still serves as good security against regressions. But how can we ensure our software works correctly in other unknown cases or on different platforms we target? It's impossible to cover every possible edge case exhaustively – this immediately multiplies the number of tests we have to execute by the number of platforms, and sometimes even more, especially in mobile platforms, where different phones from different vendors may behave differently. This produces a test oracle problem, which is costly to test every case, and a reliable test set problem, where existing test cases may not cover edge cases on specific platforms.

To alleviate these problems, a metamorphic testing (MT) approach was proposed. This approach does not need a test oracle because it can produce meaningful relations between program inputs and outputs to understand if a program contains defects. For example, such relations would be the change in performance, the amount of data collected if a test is querying for it, or some different, area-specific patterns in produced data that would indicate the presence of a metamorphic relation (MR). A metamorphic relation can be used as a test oracle for test cases. Still, instead of capturing a selection of hand-picked test cases, it captures a broader range of inputs and outputs that match the given MR [CKL+18]. The benefit of metamorphic testing was proven when it found bugs in GCC and LLVM compilers – systems that have been used and tested for decades [CKL+18].

Majority of existing examples of automated MR generation tend to search for metamorphic relations in a predictable context – either by looking for known patterns [HK18; KBB15], operate on polynomial functions to produce MRs in mathematical software [ZCH+14], or apply some basic operations on a small subset of data types to search [XTZ+24]. In practice, metamorphic relation detection is mostly a manual exercise, during which engineers with domain expertise usually determine metamorphic relations based on their knowledge and experience in a project. Such metamorphic testing strategy adds bias to the testing, where some MRs are ignored [CKL+18], additionally, such testing technique makes a pretty steep entry point, which may scare off practitioners who might not be willing to take a risk and check if MT can help in their projects, as immediate benefits are not necessarily obvious without experimentation.

Versatile automatic detection of MRs is one of the most crucial challenges in the field. Existing works, for the most part, are proof of concepts of ideas, or are pretty significantly tied to the programming language and domain – in majority of examples, the primary language is Java [ATJ+24; HK18; XTZ+24], and the methods adopted are relying on code analysis to decide the presence of MRs, thus becoming more reliant on the infrastructure of the system – compilers, optimizers, code inspection tools. Moreover, there is little investigation on MR detection during program runtime, restricting information to mostly static analysis. The primary purpose of this work will be to propose a tool capable of searching for MRs within a specific user-defined scope in a way that is system-agnostic and is capable of detecting MRs during runtime – meaning that, for example, a project written in C# could use the proposed solution just like the client project written in C++, and MRs for it are generated based on immediate results of a program under test. In such a project, the client will be expected to provide a tested function, transformations to work with when looking for MRs, and sample inputs to validate such MRs. It is believed that creating such a tool will provide insight into the viability of a universal MR generator tool. In case of success, there will be an example of reduction of initial complexity in the application of metamorphic testing, inviting more engineers to try such methodology. Moreover, by testing the said tool, new heuristics and MR patterns could be discovered, therefore broadening the field of research related to metamorphic testing.

Main goal

To propose a method of generating metamorphic relations within a user-defined scope during program run time.

Tasks

- Task 1: Perform an existing literature analysis to better understand state-of-the-art knowledge in metamorphic testing.
- Task 2: Find a set of projects that could be further analyzed and tested. Define the criteria of such a project.
- Task 3: Implement a tool to search for program-specific MRs based on user-defined transformations.
- Task 4: Generate new Test Cases based on discovered metamorphic relations for test projects.

1. Preliminaries

This section presents the theoretical basis for the upcoming work. It lists important concepts related to software testing and metamorphic testing.

1.1. Test Oracle Problem

Test Oracle is a mechanism that allows the verification of the result of a test program by a specific rule. Typically, test oracles cannot be readily determined from the test code because the expected value might only be known to the programmer who wrote the test. At the simplest example, an assertion in a unit test for function $Sum(x, y)$ may look like this:

```
AssertEquals(Sum(2, 2), 4);
```

In this case, a new oracle must be constructed if the function inputs change. Ideally, a test oracle would be derived from software specifications. Still, usually, the best oracle accessible is the human engineer who knows the expected behavior of a program under test [JCH$^+$16]. Another aspect of a test oracle problem is the complexity of constructing an effective test oracle for a system that performs unknown actions on a given input, like machine learning models [Nak17]. Defining a good test oracle might also be too expensive when testing cyber-physical systems [ATA$^+$21].

1.2. Metamorphic Testing

Metamorphic Testing relies on creating follow-up test cases from existing test cases based on a certain rule, also known as a metamorphic relation. Instead of comparing test case output to some expected correct result, as would happen in the case of unit tests, Metamorphic Testing checks the adherence to the MR by the input-output pairs generated from multiple executions of the software-under-test (SUT). This allows for a constant test oracle that will apply to new test cases generated using the MR, thus alleviating the test oracle problem. [CKL$^+$18]. Another benefit of MT is the possibility of automatically generating huge, easily verifiable test sets, thus addressing the reliable test set problem, which states that a reliable test must always reveal a defect in a program. Still, assuming that a potential set of program inputs can be infinite, such a property becomes infeasible [How76].

1.3. Metamorphic Relations

The formal definition of a metamorphic relation was defined as follows: For a given function f, a metamorphic relation is constructed over a sequence of inputs $x_1, ..., x_n$, and their corresponding outputs $f(x_1), ..., f(x_n)$, where $n >= 2$. New inputs in the sequence are called follow-up inputs and are determined from the source inputs by applying certain transformations or generator functions [SDT$^+$17].

It was discovered that in practical cases, the MR can be interpreted as a logical implication that shows that a sequence of source inputs, related outputs, and follow-up inputs implies a general relation over all inputs and outputs [SDT+17]:

$$R_i(\underset{k=1..m}{<x_k>}, \underset{j=(m+1)..n}{<x_j>}, \underset{k=1..m}{<f(x_k)>}) \implies R_o(\underset{i=1..n}{<x_i>}, \underset{i=1..n}{<f(x_i)>}) \tag{1}$$

This definition typically gets reduced to a much simpler form:

$$R_i(I_1, ..., I_n) \implies R_o(f(I_1), ..., f(I_n)) \tag{2}$$

R_i is an input relation that defines a transformation between source and follow-up inputs, and R_o is an output relation describing how source and follow-up outputs are transformed. However, in most literature, a metamorphic relation is usually considered between the source and follow-up values of a single input. This means that each separate input can be transformed differently, allowing for more verbose and broader MRs. One of the examples in the literature has proposed such a notation for one of the MRs [ATJ+24]:

$$((X_f = X_s)\&(E_f = E_s - 1)) \implies (pow(X_f, E_f) = \frac{pow(X_s, E_s)}{X_s}) \tag{3}$$

MR 3 shows that the follow-up value of X is left unchanged, and E gets modified. This conjunction implies that the equality defined in the consequent will be true. Since the function being tested here is the power function, it is easy to make a conclusion that this MR exhibits the following equation [ATJ+24]:

$$X^E = X^{E-1} * X \tag{4}$$

Metamorphic relations are the necessary properties of a program; that is, they must be logically deduced from the evaluated algorithm. This means that by analyzing the code of an algorithm, an experienced tester may be able to derive such MRs by parsing the code logic, or, in the case of mathematical or scientific software, MRs are usually directly related or identical to formulas and rules. For example, if we aim to test a library that computes trigonometry functions like $sin(x)$, we can refer to this formula to use it as a test [ZCH+14].

$$sin(\pi + x) = -sin(x) \tag{5}$$

A less formal but more readable approach to defining MRs was proposed by Segura et al. [SDT+17]. Inspired by the Goals-Question-Metric template used for software measurement, a template for metamorphic relation description was proposed. GQM addresses fields like application domain, definition context, constraints, MR name, and relations on inputs and outputs. Here's an example of such an MR definition for the distance between nodes in an undirected graph:

In the domain of $Graph\ \ Theory$

Where $Graph\ \ G\ \ is\ \ weighted\ \ and\ \ undirected$

The following metamorphic relation(s) should hold

MR_1 :

if $(x, y)\ \ -\ \ nodes\ \ in\ \ graph\ \ G$

then $(min(dist(x{,}y{,}G)) == min(dist(y{,}x{,}G)))$

The following pattern can be used as a valid formalization of MRs as requirements, therefore being a source of knowledge when creating tests. Using metamorphic relations as the specifications or as a tool for testing existing software documentation is a promising field [CT21].

Program invariants and metamorphic relations. While metamorphic relations may be considered program invariants, there is an essential difference between the two terms: metamorphic relations are applicable only in the context of program inputs and outputs that match specific relation descriptions. In contrast, a program invariant is considered to be a specific constraint that must be respected for all inputs [LSN18].

2. Literature Review

For the literature review, the primary sources were Google Scholar and ACM Library. Primarily, the search was done for publications that are at most five years old; however, in some cases, valuable insights were found in older papers, usually referenced by the newer work. Searching for specific terms like 'LLVM' in combination with metamorphic testing-related terms, for example, yielded very few relevant results. Therefore, it was decided to conduct a broader search on metamorphic testing applications across various domains. The main goal of the literature search is to uncover existing methods of automatic metamorphic relation detection. Typically, authors of such works provided the evaluation procedures of their MT methodologies and their insights on the possibility of automation and the suitability of their methods for specific software categories. This knowledge will be beneficial for further work aiming to create or improve an existing method of MR detection.

The following Research Questions were raised to ensure that the literature researched will be relevant to the defined tasks:

- RQ1: What are the existing methods in metamorphic testing?
- RQ2: What degree of automation can be achieved when searching for metamorphic relations?
- RQ3: How is the efficiency of metamorphic relations tested?
- RQ4: Software projects of which type benefited the most from metamorphic testing?

2.1. Applications of Metamorphic Testing

Since the inception of metamorphic testing as a concept over 20 years ago, much research has been done in this area, producing valuable results in computer graphics, compiler, and machine learning testing [CT21]. In most cases, metamorphic testing is evaluated in terms of scientific or purely mathematical software [AEC22]. This coincides with the main applications of metamorphic testing, where systems rely on correct and efficient mathematical libraries, which still exhibit critical bugs [DGR17]. One of the most notable applications of metamorphic testing is GraphicsFuzz – a tool for testing Android GPU drivers by applying fuzzing, random noise to existing shaders as a form of unused or redundant code, and checking if they compile and don't produce unexpected artifacts. This method has been adopted by Google in 2018 [Don19]. Like in a case with testing of GCC [LAS14], GraphicsFuzz tested how shaders are compiled and executed, meaning that the input and its follow-up was the shader code itself, and the relation between outputs was the correctness of the program compiled. In such cases, simple code mutation is enough to produce new follow-up inputs. Another example of MT applications in the industry is the Metamorphic Interaction Automation (MIA), developed at Facebook [FB], where MT was used to test a system that simulates Facebook's platforms within an enclosed network – a highly non-deterministic system, with a considerable degree of test flakiness – unstable test which, over series of runs on the same code changeset, exhibits both successful and failing test runs. The metamorphic testing approach helped engineers at Facebook uncover certain relations between simulated users in offline simulations, where flakiness was absent, and online simulations that suffered from instabilities.

Although the usage of metamorphic testing was not automated, and MRs were defined manually [FB], it's still a curious case of the application of MR in the industry.

Metamorphic Testing relation with fuzzy testing. The term "fuzzy testing" is often confused with Metamorphic Testing. However, these testing approaches are different. Fuzzy tests involve making random changes to static data, such as program code or an image, and then checking how the program reacts to the increased chaos in the input. The core aspect of fuzzy tests is that chaos gets introduced without any patterns or logic in such transformations, thus preventing the making of assumptions about potential changes in the output. The practical use of fuzzy testing involves checking for typically binary conditions: does a program compile? Or does it crash when compiled and executed? Does the image get correctly classified with the added noise? Metamorphic relations are capable of answering more in-depth questions about the program state, though they require more effort to be produced.

2.2. Existing Metamorphic Testing Methodologies

The effectiveness of metamorphic testing relies heavily on the quality of metamorphic relations detected. Therefore, the main focus of the research is usually on the detection, definition, and location of metamorphic relations. Improvements in the detection or generation of MRs are crucial because this component in a metamorphic testing pipeline cannot be fully automated using existing techniques. Automating this process is vital for several reasons: First, in most practical cases, metamorphic relations are detected manually by experienced engineers who know the domain under test [CKL+18]. This increases the cost of such testing methods significantly. Secondly, and most importantly, it creates a more significant possibility of human-error-related problems, where sure MRs might be ignored due to subjective reasons [XTZ+24].

The following is an analysis of existing methods for detecting or generating metamorphic relations. Even completely different approaches are expected to have some common fundamental obstacles that need to be resolved, such as testing data preparation and final method efficiency evaluation. Hopefully, the analysis of several different approaches will help answer these questions.

2.2.1. Metamorphic Relation Detection Using Machine Learning

The most straightforward approach – defining several patterns resembling a metamorphic relation and using them in the machine learning-based classifier, was proposed by Kanewala et al. This approach revolves around producing a Control Flow Graph (CFG) from the given source code and looking for metamorphic relation patterns in the nodes of the directed graph [KB13]. The patterns used for this method were initially defined by Murphy et al. [MKH+08], and mostly are valuable for mathematical functions or functions that handle data structures like arrays (Table 1).

CFG was created with nodes that contained only atomic operations. Such nodes have simple semantics, allowing them to label them as small operations like $assignment$, $addition$, $goto$. These labels were later used for further supervised learning of an MR prediction model. Sequences of such

Metamorphic Relation	Transformation of the Input
Additive	Add or subtract a constant
Multiplicative	Multiply by a constant
Permutative	Randomly permute the elements
Invertive	Take the inverse of each element
Inclusive	Add a new element
Exclusive	Remove an element
Compositional	Combining two or more inputs

Table 1. Base metamorphic relations

labels may suggest the presence of a metamorphic relation, which then gets used to produce new test cases. The method's validity was tested on 48 functions, ranging from purely mathematical to operations with arrays. The solution's effectiveness was evaluated using mutation analysis, and the faults were only injected into the tested method. Initial test cases for each function under test were created randomly, and then follow-up test cases were produced based on detected metamorphic relations. At its best, the proposed method was able to capture 66% of the mutants. However, it was noticed that some mutations could not be killed because the final output of a program remained the same.

Control Flow Graph Post Processing. The original work [KB13] extracted two types of features from the generated CFGs – Node and Path features. Node features are depicted as $label - inputDegree - outputDegree$, while path features are constructed as a short sequence of operation labels from start to exit. Such differentiation is significant because, in some cases, permutative metamorphic relations could have been detected only when the sequence of operations was known. The next iteration of this method proposed an improvement to the handling of CFGs, where features are extracted using graph kernels [KBB15]. A graph kernel is a method of comparing two graphs. Two different graph kernels were analyzed – the random walk kernel and the graphlet kernel. Random walk kernel combines the values of the transitions between nodes in each path of a graph and then compares the results of both graphs, while graphlet kernel constructs smaller subgraphs with sizes 3, 4, and 5 and compares the count of matching subgraphs in these graphlets.

Model accuracy was measured for every vector produced. The evaluation procedure relied on getting two values – balanced success rate (BSR) and the AUC, the area under the receiver operating characteristic (ROC) curve.

BSR is calculated using the following formula:

$$BSR = \frac{1}{2}(P(success|+) + P(succeess|-)) \tag{6}$$

$P(success|+)$ is the probability of correct positive classification and $P(success|-)$ is the probability of correct negative classification.

However, the primary evaluation method was AUC, which was more effective in measuring learning algorithms [KBB15]. AUC is a way of determining the probability of a model correctly predicting true positive and true negative classes, which is computed as the area under the curve produced by plotting values for true positive and false positive rates (TPR; FPR).

$$TPR = \frac{TP}{TP + FN} \tag{7}$$

$$FPR = \frac{FP}{FP + TN} \tag{8}$$

It was concluded that the support vector machine (SVM) model is a significantly more effective algorithm than the decision trees, with AUC being mostly above 0.8 in the case of SVMs [KB13]. After testing both graph kernels and previous approaches of extracting node and path features, it was concluded that the random walk kernel combined with the SVM algorithm is the most effective method of searching for metamorphic relations. For most metamorphic relation types, the BSR and AUC values were the highest for the random walk kernel method. In contrast, the graphlet kernel was significantly worse than the original approach of extracting node and path features. Excluding invertive metamorphic relations, for which AUC scored 0.74, other MR classifiers had $AUC >= 0.8$ – a value that suggests that the created classifier is good.

The metamorphic relation detection method, proposed by Kanewala et al. [KB13; KBB15], has shown that the strategy of analyzing underlying source code using machine learning-based classifiers can be an effective method of detecting applicable metamorphic relations. However, such a method can only be used if the MRs being searched are adequate for the area under test. In this work, testing was done on code projects specializing in machine learning and scientific solutions. Moreover, detected MRs won't be automatically tested, so a follow-up process of test generation and validation is needed to use detected MRs. However, the main limitation of this method is that it can detect only one MR per label, even though, practically, a single label may exhibit multiple MRs. This problem was addressed in another method that utilized multi-label neural networks for MR detection in the same projects, and it was proven that using more labels can detect more MRs [ZZP+17].

Training Data Generation. Machine learning models typically require a lot of data samples for training, which adds a significant level of complexity when using such techniques. One of the alternatives to searching for relevant open-source software projects for samples, a synthetic data generation method, may be helpful. Such an approach was tried by generating mutated code from the original samples and using them for training while keeping the SVM and graph kernel algorithms. Unfortunately, such a strategy was not helpful in all cases – most mutants decreased the final ROC values. Only mutants that changed the atomic operations and preserved the code's final logic helped increase the ROC value [NME19]. The explanation for this behavior was provided in the follow-up work, which stated that if a program contains the MR, its semantically equivalent variants will also exhibit the same MR. As for the non-equivalent mutants, the syntax might be nearly identical, but a single logic-changing mutation will completely change the output of a program, thus destroying the MR [Göt23].

Natural Language Processing. Another method of acquiring metamorphic relations from the source code is code comments and documentation analysis. For example, JavaDoc – a typical source of auto-generated Java source code documentation, can contain essential properties of the code, such as invariants and constraints, or an abstract description of the logic, which provides valuable insights for MR inference, like the statement that the developed function in some instances should behave like a function from the standard library [BGE+21]. Such MRs are natural to obtain and easy to formulate further by a tester, but processing them automatically poses a natural language processing problem. "MeMo", a project that analyses Javadoc's comments written in natural language, has proven the effectiveness of such an approach. After analyzing over 7000 sentences across several core Java projects specializing in scientific software, authors of "MeMo" have constructed a model capable of detecting MRs not observed by search-based methods [BGE+21]. A more straightforward strategy was adopted in the "MRpredt" project, where specific keywords in the Javadoc were focused, like *@param* and *@return* [RKK20]. However, compared with the base "MRpred" tool proposed by Kanewala et al., which analyzed the source code directly, [HK18], its variation for Javadoc analysis only improved results for a fraction of MRs detected.

The strategy of selecting the best metamorphic relation in an automated way was implemented with a reinforcement learning technique known as context bandit. This method was called Adaptive metamorphic testing (AMT) [SG20]. It accepts the target program or algorithm, a test suite, and a set of applicable MRs. The contextual bandit evaluates each MR by constructing new follow-up test cases based on a test suite. If, for a given MR, it was proven that follow-up test cases are not adhering to the MR, then the algorithm is considered faulty, and the detected MR is then used as input data for the subsequent search iterations, thus removing contradicting MRs. Such a strategy allows the selection of the best metamorphic relations while remaining independent of the application domain. This is especially useful because not every typical MR benefits test generation. In most cases, the only way to tell if MR is helpful is to evaluate it manually. Adaptive metamorphic testing allows for automation of this step. Experiments on image recognition have shown that such a strategy tends to be more time-efficient and effective than selecting MRs randomly or exhaustively going through each possible MR [SG20].

All in all, metamorphic relation detection using machine learning is a promising area. However, very little testing outside of purely scientific software, as well as a lack of high-quality datasets, raises questions about the validity of this method for a broader scope. Since the technique relies heavily on having examples of good MRs, experiments with Machine Learning have produced a significant amount of well-defined standard metamorphic relations or input transformations, which are used in other works, unrelated to ML-based techniques [ATJ+24; ZCH+14].

2.2.2. Test Case Based Metamorphic Relations

Another approach to metamorphic relation detection is based on the idea that initial test cases, such as unit or integration tests, implicitly contain information about possible MRs. One such method was proposed in an application called MR-Scout [XTZ+24] – a tool capable of detecting metamorphic relations in tests for object-oriented programs written in Java.

MTCs, or MR-encoded test cases, rely on the idea that metamorphic relations are formed between at least two inputs and corresponding follow-up outputs directly related to the initial inputs. Consider the following pseudo-code resembling the test of a stack data type:

```
PushPopTest()
{
    let a = stack();
    a.push(42);
    let b = stack(a);
    let firstStackResult = a.pop();
    let result = b.pop();
    assert(result = firstStackResult);
}
```

Listing 1: Example of a test for a ”Stack” class

In this case, an instance of a *stack* data type *a* and an initial value of 42 forms two source inputs to stack *b*. The result of *b.pop*() is a follow-up output based on the initial input, and it must be 42. Therefore, this code contains the following metamorphic relation[XTZ+24]:

```
    x = stack.push(x).pop()
```

Listing 2: metamorphic relation in code

The importance of uncovering this MR is shown by the possibility of using the following rule in creating a new Metamorphic Test, which can be parameterized:

```
PushPopTest_MT(x: int)
{
    let a = stack();
    stack.push(x);
    assert(x = stack.pop());
}
```

Listing 3: MT based test case

To automatically produce new test cases, the MR-Scout tool consists of three phases:

MTC Discovery. Since encoded MRs cannot be deduced syntactically, the discovery of MR-encoded test cases relies on detecting two properties that would suggest the existence of an MR. The first property is called **Method Invocations**. It verifies that at least two method invocations of the same object have two separate inputs. Another property is called **Relation Assertion**, and it checks if a test case contains at least one assertion, checking the relation between inputs and outputs of the method invocation [XTZ+24].

MR Synthesis. Synthesizing metamorphic relations involves dedicating components of an encoded MR and codifying these components into an executable method, creating a parameterized unit test (PUT). However, the information collected at the MTC discovery phase only covers follow-up outputs and no information about initial inputs. MR Scout searches for a collection of MR components, such as the target method, source and follow-up input, input transformation, source and follow-up outputs, and the final output relation assertion.

According to the MR synthesis logic, in case of $PushPopTest$ mentioned above, target methods would be $stack.push()$ and $stack.pop()$, source inputs would be stack a and constant 42, follow-up input is stack b. Source output is $firstStackResult$ and follow-up output is $result$. The only transformation is the initial transformation $a.push(42)$, which modifies the source input a. Final relation assertion happens on source and follow-up outputs [XTZ+24].

MR Filtering. Newly created test cases might not be valid metamorphic tests because of possible crashes or exceptions caused by the input. Therefore, the input that triggers such an outcome must be filtered out, as it is considered invalid. However, only having such validation is not enough to remove invalid inputs altogether, as developers might have checks and validation for invalid data, thus not causing fatal errors or exceptions. In such cases, false alarms might start appearing in the form of errors within the assert call. Authors of MR Scout consider the MR that produces such input faulty and filter it away [XTZ+24].

Method Evaluation. The MR Scout tool was tested across 701 open-source Java projects, and over 11000 MR-encoded test cases were detected, each containing around 1.9 metamorphic relations on average. The most significant MRs (64%) were based on leveraging two method invocations, while 72% of all MRs had no input transformations. The validity of the detected MRs was checked manually. 164 MR samples were selected randomly, and the detected MR was checked to see if it was true. Authors of MR Scout uncovered a fault in their tool, which caused false positives when validating the Assertion Relation parameter. This condition was incorrectly triggered on re-assigned variables. However, only 4 cases exhibited such behavior. Unfortunately, automated ways of detecting such faults were not proposed. Inputs for codified MR testing were produced using EvoSuite, an automated test case generation tool in Java. Later, the effectiveness of newly created test cases was checked using mutation testing techniques [XTZ+24].

The benefit of deriving MRs from automated tests is twofold – first of all, MRs allow the construction of parameterized tests, thus reducing maintenance costs of tests and improving potential coverage. Secondly, MRs can serve as a specification of how inputs to the code get transformed, providing a more formal way of understanding the behavior of a system, which could help evaluate the overall validity of a test. Additionally, deriving MRs from unit tests may be a good start for domain-specific MR search when base metamorphic relations defined by Murphy [MKH+08] may not be enough to uncover defects in a code. The examples provided by the authors of MR Scout are done for object-oriented programming, where methods of a single instance of a class are tested, thus allowing the inclusion of the effect of the changed state of an object. The efficiency of applying a similar MR detection strategy in non-object-oriented programming projects was not evaluated, and similar projects that would tackle this were not found.

2.2.3. Specification Based Metamorphic Relations

As discussed previously, important observations about MRs encoded in the program code can be obtained from analyzing JavaDoc comments or by parsing automated tests, which typically check for behavior defined in such specifications, thus encoding MRs in themselves. These observations suggest an intuitive conclusion that a metamorphic relation represents an adherence to a particular specification of the software if the written code is valid in terms of software requirements. This, in turn, means that a specification document is a good source for metamorphic relations that are either explicitly or implicitly encoded in the program [SFP+21].

A widely used method of extracting test cases from software specifications is the category-partition method (CPM) [CR99]. CPM defines a process of splitting the given functional specification of a program into specific system state descriptors known as categories, then further dividing each category into a collection of choices or a frame – a set of values possible in a category, and defining the constraints between different categories, thus formulating a basis for a test specification. A test specification can then create test cases by applying the possible values for a given choice [SDL+23].

Based on this knowledge, a methodology to extract MRs from encoded specification was proposed in a tool called METRIC [SFP+21]. Authors of these tools have shown that a frame can be used to define the MR, as it corresponds to a collection of test cases with a common constraint or a property. Since the frame already contains the description of relations between inputs, the MR detection problem turns into an issue of validation that outputs generated for given inputs also have a relation encoded between them. The system proposed by METRIC is following:

- Select two full frames B_1 and B_2, that is, frames that define values for all choices.
- By definition of CPM, B_1 and B_2 must contain different choices.
- The difference between choices will specify the expected difference between them according to test case outputs. Combined, relations between input and output pairs form a metamorphic relation. If, after execution of given test cases, the difference between outputs is not as expected, the program is considered faulty.

It is important to note that this method is designed to systematize but not automate the detection of metamorphic relations. Instead of searching for MRs in an ad-hoc manner, where an engineer would manually search for possible relations between inputs and corresponding outputs, METRIC allows to narrow down the search by utilizing existing specifications that describe how a change in one or several constituents of a test case will affect the final output of a program. The effectiveness of this methodology was tested practically by asking engineers with basic knowledge of testing to formulate MRs for given software projects. It was shown that this methodology made MRs far more accessible to detect than doing the same work by trying to deduce the MR [SFP+21].

Data mutation techniques. An approach similar to MT was proposed for testing software that operates on structurally complex inputs, like XML parsers and compilers. Instead of modifying the program under test to capture mutants, a test itself was suggested to be modified based on some rule derived from existing tests or software specification, thus producing new test cases based on such mutants [SZ09]. Such rules are called data mutation operators (DMO). The data mutation approach naturally comes in comparison with metamorphic testing performed on similar principles. The main difference between data mutation testing and metamorphic testing is that MT allows us to automatically verify the validity of new test cases by checking metamorphic relations. At the same time, DM requires manual validation [Zhu15]. Similarities between DM and MT have inspired the creation of μ-MT. This metamorphic relation detector applies data mutation rules to construct related test inputs, thus implicitly defining the antecedent in the Equation (2). The output relations are then determined manually by inspecting the potential ties between produced outputs and referring to the expert knowledge [SJW+23].

2.2.4. search-based Software Engineering (SBSE) Solutions for Metamorphic Relation Detection

Previously listed methodologies assumed the existence of a metamorphic relation in the code and were searching for patterns, properties, or heuristics that would indicate the presence of an MR, such as known metamorphic relation in a CFG [HK18; KBB15] or a presence of typical code elements [XTZ+24]. These methods rely on having access to the source code (white box testing) and offloading the duty of generating inputs for MRs to other tools in the testing pipeline. But what if no initial MRs are known, or only basic MRs can be detected? In such cases, search-based software engineering generates and tests new MRs. Similar methods were also tried when searching for program invariants, generating classical test cases, and performing regression testing[GC08]. Since MRs can be considered a subset of program invariants, adopting SBSE methods for MR generation is a logical approach.

A strategy for inferring metamorphic relations through metaheuristic search algorithms, such as particle swarm optimization, was proposed by Zhang in a project called "Metamorphic Relation Inferrer," or MRI. Out of 70 MRs evaluated across multiple sources, the authors of MRI concluded that 61% of evaluated MRs are polynomial [ZCH+14]; therefore, searching for such MRs can bring the most value. Polynomial MRs define relations between all inputs and all corresponding outputs as polynomial equations. A typical example of such MR is a well-known property of a sine function, as described in Formula 5.

The metamorphic relation between input x_1 and follow-up input x_2 that would satisfy Equation (5) can be constructed using the following linear equation:

$$x_i = \sum_{j=1}^{n} a_{nj} x_j + b_n \tag{9}$$

For values of a_{11} and b_1 as 1, the follow-up input satisfies the MR. This means that the inference of new metamorphic relations can be seen as a search for linear equation parameters or a higher degree polynomial. MRI searches for a vector of parameters for a polynomial using the Particle Swarm Optimisation algorithm. The algorithm generates N candidate solutions, known as particles. Each particle's velocity and location are described, and these values are stored in a D-dimensional space with N particles. The velocity and location of each particle are then computed based on the results of other particles, preferring results that are getting closer to the target value. PSO algorithm makes minimal assumptions about the problem, which does not guarantee that an ideal solution can be found. However, this allowed authors of the MRI to select weights of the algorithm based on existing experiments with PSO, done outside of metamorphic testing[ZCH+14]. However, the MRI was shown to only handle trigonometry functions with well-known base metamorphic relations – the true mathematical equation for such functions. Unfortunately, the efficiency of this tool in other areas or different mathematical functions could not be evaluated.

Genetic programming could also be used as a tool for detecting MRs. GenMorph discovers numerical, boolean, and ordered sequence-based MRs in Java code. [ATJ+24]. This tool uses an evolutionary algorithm to search for possible best representations of MRs for function implementation. The search for MRs drives rewards for the algorithm with as few false positives (FP) and false negatives (FN). It is important to define FPs and FNs in terms of metamorphic relations. Authors of GenMorph describe them as follows:

A False Positive of a metamorphic oracle. When a pair of inputs x_1 and x_2 for program $P(x)$ are correct, but output relation $R_o(P(x_1), P(x_2))$ is negative, therefore $R_i(x_1, x_2) => R_o(P(x_1), P(x_2))$ is $false$.

A False Negative of a metamorphic oracle. When inputs for an incorrect version $p(x)$ of a program $P(x)$ accepts the pair of inputs, and $R_i(x_1, x_2) => R_o(p(x_1), p(x_2))$ is $true$. An MR which accepts false negatives cannot be used to expose mutants in the code.

GenMorph utilizes automatic test case and mutation generation tools for source input preparation and procures required input transformations based on known patterns that represent MRs. [ATJ+24]. Authors refer to such base MRs as canonical MRs. A follow-up input is computed based on MR transformation for each source input. These input pairs produce source and follow-up outputs for both source codes. It's mutated versions, thus generating an array of tuples consisting of inputs and outputs for each generated test case. Then, an evolutionary algorithm is applied to the created data set, searching for new relations between inputs and corresponding outputs: two separate populations are produced from the initial canonical MR, and they are evolving in parallel, competing to reach better metrics for FPs and FNs, and in reaching the smallest possible MR, while having the same semantics. The tool was tested on various software applications that operated on numeric, boolean, and sequence values, such as arrays and strings. As a standalone testing tool, GenMorph was not able to outperform test generators like Randoop or Evosuite, but some mutations were only uncovered by the metamorphic tests, that is, combining both generated tests and produced MRs have increased the amount of mutants killed to existing mutants ration, also known as the mutation score.

As a strategy for filtering out MRs with false positives or false negatives, authors used OASIS [JCH+16]. This tool checks the test oracle by providing inputs that would cause the false negative to appear. If the false negative is found, the oracle, or the MR, is considered faulty and will be removed from the tested population. OASIS uses the common tactic of generating mutants when searching for false positives or false negatives in the MR. In case of false positives, OASIS searches for assertions within the tested code and inverts the checks. If a test is still passing, such an oracle is prone to generate false positives and needs to be changed. The tool can provide possible improvements to the oracle, but this approach is unreliable enough to be used automatically. Human intervention is still required to verify if a proposal is correct from a program specification perspective. Although OASIS is Java-centric, it once more validates the use of mutation testing for MR validation. Additionally, authors of OASIS propose valuable insight into test oracle improvements: If a test oracle is more likely to produce inaccurate results, such as false positives, it can be improved if it's negated [JCH+16].

SBSE-based metamorphic testing has found its use cases in industrial applications where quality or performance metrics might be critical. For example, a tool called GAssert-MR was tested on the elevator management system functioning in a simulation, which computes the best routes and work distribution between multiple elevators. Apart from functional behavior, tests for such systems care about nonfunctional aspects such as the number of engine starts, total distance traveled, and energy used [ATA+21]. Since the automated elevator service functions in a very dynamic environment, where a slight change of a property of an elevator call dramatically affects the final behavior, GAssert-MR addresses the change in these metrics indirectly by modifying existing test cases by adding extra random passengers, changing the number of available elevators and tweaking the start position of each elevator. Significant and unexplained deviations from the expected quality characteristics after a test run indicate a potential functional failure. This shows that metamorphic testing can be cost-effective for cyber-physical systems, where generating precise test data is very expensive [ASA+20; ATA+21].

Improvements in composite MR generation. One of the improvements of metamorphic testing methodology has proposed the concept of composite MRs [LLC12]. Such metamorphic relations are built with the idea that if source and follow-up test cases for some MR_1 can always be used as source test cases for MR_2, these two MRs are composable. This applies to more MRs; if another MR can be added to the existing composition, it is possible to construct k-composite

MRs. The benefit of such an approach is that it allows the creation of a multi-property test oracle that is capable of evaluating several qualities of an SUT in fewer test runs, and it was proven that k-composite MRs have the same effectiveness in detecting failures as each MR executed separately [LLC12]; The problem with this approach is that it requires substantial knowledge and additional manual effort when constructing such MRs, especially having in mind that MR detection is a process that is affected by human biases, which sometimes causes important MRs to be ignored. Another crucial property of composite MRs is that this composition is not commutative. While it might be possible to compose MR_1 with MR_2, it may not be possible to do it in reverse. [XWY19] The problem of manual MR composition was addressed by utilizing a genetic algorithm. Each individual is a collection of MRs that form a valid composite MR (CMR); these individuals are crossed together to reach the best composite MR, which would combine as many MRs as possible by creating the most extended sequence of composited MRs. However, always going for the longest composite MR is not cost-effective because creating such large CMRs does not achieve better mutation scores than shorter CMRs. In the case of the trigonometry functions tested, the best performer was a composite MR constructed from 3 separate MRs, while longer CMRs produced worse or similar results [XWY19]. Using a genetic algorithm has helped reduce the problem space by filtering out poorly performing MRs and building on the compositions that achieved better mutation scores. Performing a brute-force composition would produce more noise and would be slower. Authors of this paper also propose a practical way of calculating the mutation score for a follow-up test case FT produced by the MR:

$$MS(mr_i, FT) = \frac{N_i}{N_p - N_e} \tag{10}$$

N_i is the number of mutants killed, N_p is the total number of mutants, and N_e is the number of equivalent mutants, which produce the same semantics as the original code. This equation adjusts for the equivalent mutants, allowing us to obtain more precise results. The results of the experiments conducted by the CMR authors have confirmed that composite MRs reliably increase fault detection capabilities.

Search-based methods for metamorphic relations detection are an up-and-coming area. They allow the discovery of new MRs specific to the code being tested while also providing quite a high degree of automation, even though they are still incapable of being fully automated. Not providing good initial MRs to iterate by searching for new generations of MRs from will make the algorithm search for an extensive set of possible outcomes, thus taking a potentially vast amount of time without providing meaningful results. Therefore, good initial MRs need to be parameterized and optimized further.

2.3. Summary of methods

The methodologies found were listed in terms of their degree of automation – how much of the manual work is needed to get the final MR for a given program. Table 2 contains a compilation of methodologies and their respective descriptions.

Method	Degree of automation	Main points
Machine Learning-based classification	Partial	Shows the presence of known MRs in the program but does not produce new MRs.
Natural Language Processing of automated docs	Partial	Very large amounts of data need to be produced and analyzed in order to properly handle subjective differences in the natural language.
Reinforcement Learning-based MR detection	Partial	Can automatically find the best MR for a program from a set of predefined MRs, but is unable to produce new MRs.
Test case-encoded MRs	Full	Capable of detecting MRs in tests and building new test cases automatically. After manual testing, some MRs were found to be incorrect.
Software specification-based MR detection	Manual	Systematizes the process of manual MR detection and improves the quality of MRs found manually.
Search-Based Software Engineering methods	Full	Can produce completely new MRs from a set of inputs, related outputs and their transformations. Primarily adopted in scientific software testing.

Table 2. MR detection methodologies

3. Feasibility Study for Metamorphic Relation Detection Techniques

In the majority of the literature found, the tooling for test case generation, mutation testing, oracle verification, and other related tools was applicable in the context of the Java programming language environment [ATJ+24; HK18; XTZ+24]. In this chapter, a short feasibility study was conducted to ensure the primary goal of the future work – automated detection of MRs in the context of programming languages like C++ (Clang).

The simple prototype of the MR constructor was written in C++. Based on the brute-force approach, a small selection of input and output relations was conducted, assuming that only one extra operation, chosen from four basic arithmetic operations, can be added to the source value to produce the follow-up value. The algorithm first sets the input relation and then tries to find the output relation closest to the outputs produced by the follow-up inputs by trying every combination of operations on the follow-up output and inputs.

The fitness of the results is computed using the cosine similarity formula, which calculates the distance between two vectors:

$$S_C(A, B) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{11}$$

However, cosine similarity is not a reliable proof that two vectors are identical, so for close vectors, an extra check is done that compares if the data sets are the same. The function under test was the $pow()$ function, which accepts two arguments: x and e. The algorithm constructed was able to find the MR, for which the cosine similarity was closest to 1, and later checks provided that the data vectors are identical:

$$(R_i((X_f = X_s)\&(E_f = E_s - 1))) \implies (pow(X_f, E_f) = \frac{pow(X_s, E_s)}{X_s}) \tag{12}$$

This MR is also a valid MR found for $pow()$ in the case study proposed by the authors of GenMorph [ATJ+24]. This outcome was expected because the prototype and its testing environment are inspired by the approach proposed in GenMorph. However, this simplified version used cosine similarity to guide the best MR instead of collecting false positives and negatives. Even though it's very simple, this example algorithm already gives an intuition about the possible scope of computation needed to capture more complex MRs, involving more inputs and possible transformations, and the importance of adopting solutions that would help reduce the required number of iterations. The proposed algorithm is listed below, and the link to the source code can be found in the appendices.

Algorithm 1. Algorithm for simple MR generation

```
procedure MR Generation for program P(X)
    transformConstants ← Array of constants to be used in input transformations
    outputTransformations ← Array of operations to be used in output relation generation
    canonicalMRs ← Array of base MRs to derive new relations from
    inputs ← Randomly generated array of inputs
    outputs ← Array of outputs for original inputs
    followUpOutputs ← Array of outputs for transformed inputs
    searchedOutputs ← Dictionary of arrays containing transformed outputs
    createdMRs ←Array of created MRs
    for each $c \in canonicalMRs$ do
        for each $x \in transformConstants$ do
            $outputs$ ←[]
            $followUpOutputs$ ←[]
            for each $input \in inputs$ do
                $transformedInput$ ←$c(input, x)$
                $outputs.add(P(input))$
                $followUpOutput$ ←$P(transformedInput)$
                $followUpOutputs.add(followUpOutput)$
                for each $t \in outputTransformations$ do
                    $searchedOutputs[t].add(t(followUpOutput, input))$
            $maxCosineSimilarity$ ←0
            $bestOutputTransformation$ ←0
            for each $t \in outputTransformations$ do
                $cosineSimilarity$ ←$CosineSim(searchedOutputs[t], followUpOutputs)$
                if $cosineSimilarity > maxCosineSimilarity$ then
                    $maxCosineSimilarity$ ←$cosineSimilarity$
                    $bestOutputTransformation$ ←$t$
            $StoreMR(createdMRs, c, x, bestOutputTransformation)$
```

4. Summary of Literature Review

The majority of research in the field of metamorphic testing is conducted in the area of MR detection and formulation. This can be explained by the fact that MRs can be used as test oracles, the constructs for the expected values during testing. Once a test oracle is present, testing becomes automatable, just like the typical tests with predefined oracles. This work has produced a summary of a collection of methods, ranging from a manual process that adheres to some framework to applying genetic algorithms to create new MRs from existing ones in a semi-automated way, and conclusions on them that will be useful for further work.

State of the art. For a metamorphic testing strategy to be used, it must be cost-effective regarding the effort and computing resources required. The leading contenders are SBSE-based methods, which utilize meta-heuristic genetic algorithms that are capable of discovering the best results over time [ATJ$^+$24; ZCH$^+$14], and reinforcement learning methods working on the similar principle of improving the performance of MR detection across generations [SG21]. Unfortunately, the effectiveness of these solutions cannot be compared, as the case studies proposed were very different – while SBSE solutions improved mathematical software testing, the reinforcement learning approach was tested on computer vision software. However, both techniques are significantly more efficient than purely random or manual testing approaches. It is also essential to add that search-based software engineering methodology was not adopted for metamorphic testing only; it has served as a valuable method for test case generation outside of MT and was proven to be, albeit generic, but effective method for new test case inference [GC08; McM11].

Automation friendliness. MT does not require a predefined test oracle. Contrary to data mutation techniques, this feature automatically produces and executes new test cases. However, due to the enormous space of possible MRs in each program under test, the risk of capturing useless MRs is also considerable, which may cause inefficient use of computing power. It is also not likely to rely on a default set of MRs for each program due to the very tight relation with the area of software being tested, albeit the effective MRs for the specific problem can indeed be converted into more abstract MR patterns and be used later. Due to this problem, it is suggested to capture initial, good MRs manually by utilizing frameworks like METRIC for specification-based MR detection and then build new metamorphic relations based on these findings [ATJ$^+$24; XWY19; ZCH$^+$14]. A genuinely efficient autonomous MR detection procedure has not yet been proposed. However, the lack of a universal fully automated MR generator might not be a significant problem, as there are frameworks and methodologies for manual MR detection that have proven effective and easy to use by beginners. The focus of the research is usually on building new MRs based on existing ones because this problem is more critical in the current state of MT.

The variety of case studies. The substantial amount of analyzed methods for MR detection ([ATJ$^+$24; HK18; ZCH$^+$14]) were tested only on base MRs applicable for mathematical and scientific functions, defined by Murphy [MKH+08]. This raises some concerns about the effectiveness of these methods outside of the scope of purely mathematical software. Conversely, it is intuitively expected that a limited amount of MRs can be applied to a broader scope of programs. Different fields that were used as case studies were computer vision [SG21], network simulations [FB], and cyber-physical systems for controlling automated elevators [ATA+21], and MRs defined by the authors of these works were not specific to mathematical libraries. GenMorph [ATJ+24] has addressed the 'sequence' types like Java Strings. Still, the concern is that SBSE methods may be problematic when adapting to a broader range of nonnumerical inputs, especially for custom data types.

5. Proposed MR generation method

Practicality of search-based MR generation. Existing proposals for MR generation, as discussed in the literature review, are very tightly related to a specific language environment and tooling like automatic documentation generators, test case generators, or specific language syntax that a model is trained to read [ATJ+24; CR99; KBB15]. Proposed search-based methods only focus on one particular domain or a use case. This restricts software engineering practitioners from experimenting with metamorphic testing by drastically increasing the cost of this approach. For various use cases, an individual has to create a set of tools that may be more generic, suggesting a tooling that can be reused. This work aims to propose a strategy for such a universal tool that would reduce the cost of entry into metamorphic testing, attracting more interest to this testing strategy and allowing it to expand to more specialized usage scenarios by starting from a basic yet fundamental set of tools. While metamorphic relations can be inferred by following some heuristics and manual analysis of code, automatically generated metamorphic relations could still provide perspective on a program from angles not considered by an engineer. Moreover, not every engineer, even within an organization, knows the tested system deeply enough, making metamorphic testing even less appealing since its efficiency becomes harder to measure.

MR search process for a selected function. As shown in the prototype, it is feasible to manufacture metamorphic relations automatically by combining possible transformations and arguments with relevant output transformations that match the defined input-output relation. The proposed work proposes a strategy for MR generation capable of detecting metamorphic relations in a provided abstract function, executed in a black-box manner – the caller does not need access to the code of the tested API, allowing for direct on-platform testing, if such needs arise.

Aya is written in C++26 using template meta-programming techniques to enable support for functions with arbitrary signatures. Moving to templates had an impact on the final size of the library. It also increased complexity in inter-language compatibility compared to the original code in C, which only targeted MR generation for simple functions, as shown in Listing 4.

```
void func(void*, void*, void*)
```

Listing 4: Tested Function Signature

Function results are packed into *states*. A *state* is a vector of arbitrary types, containing the function output alongside references to passed arguments. Keeping references to passed arguments is essential because, in some cases, a function may modify passed values too, or if a function has no return value, then the state change may only be detected by comparing initial and follow-up arguments. Here is an illustrative example of *states*. Consider Function 5:

```
double pow(double, double)
```

Listing 5: cmath pow() function

An initial state for inputs $[2, 3]$ would be $[8, 2, 3]$. The output comes first because it makes it more efficient for template code to access it in post-processing. State generation makes it straightforward in further MR generation. First, initial states are produced with passed sample inputs, then are altered using provided transformers, producing new follow-up states with modified inputs. Later, such $state$ vectors are compared predictably. Originally, the support for abstract data types was provided using a void pointer alongside the data size variable. This has severely increased the risk of memory bugs, reducing the confidence in the tool for more complex scenarios. Adequate coverage for the data types covered was achieved by using the $std :: any$ class. This is a type-safe container for most data types in C++.

Analogous to the tested function, transformers defined using templates for a void function, but with arbitrary arguments, at least one of which is a reference. Such transformer functions can be applied to a given value in any order and scale, forming so called transformer chains, applied as shown in Listing 6

```
   Add(inputState[2], 3),
   Sin(inputState[0]),
   Sub(inputState[1], 10),
   ...
```

Listing 6: Tested Function Signature

To allow for combining transformers with different signatures, an $ITransformer$ interface was designed to apply such transformers in a polymorphic manner. Usage of templates and type-safe alternatives to void pointers has made the MR generation for C++ code quite universal, requiring the user only to provide the test data and generate MR generator code for their specific needs. Omitting sample data generation, transformer function preparation, and other things that users might want to do specifically in their use-cases, a call to Aya MR generation procedure consists of the following calls:

```
// Transformers with a signature void(double&, double)
std::vector<std::shared_ptr<Aya::ITransformer>> transformers =
   Aya::TransformBuilder<double, double>().GetTransformers(functionVector,
      functionNameVector);
// Tested function double pow(double, double)
auto mrBuilder = Aya::MRBuilder<double, double, double>(testedFunction,
   comparerFunction, transformers, <flags>, ...);
mrBuilder.SearchForMRs(testedInputs, <flags>, finalMRs);
```

Listing 7: Core functions in the API

For complete examples of the MR generation procedure, please refer to the source code listed in the Appendices.

Native and managed code interfaces. Interaction between *Aya* library written in C++ and a client implemented in a language is C# is possible due to *cdecl* calling convention. This helps improve interoperability between different programming systems since managed languages like C# can have a native bridge with C++ code like Platform Invoke [jko].

5.1. API design

Existing MR generation solutions [ZCH+14] focused on synthesizing metamorphic relations for simple scalar data types like integers, with complex data types like sequences being skipped or partially implemented. The logic behind such a strategy is understandable – it's easier to focus on a single data type, and it's more efficient to focus on scalar data types since they are more important for numerical software. However, creating a program that would work with more data types efficiently is harder, since there are an infinite number of such data types. Additionally, a typical search for MR involves evaluating the state of a program. For mathematical software, the final state of a tested function is a vector of scalars that is easy to compare against using standard comparison mechanisms, while in case of more specific data structures or business logic, it is harder to reason about metamorphic relations in terms of tracked states.

The creation of tools that are usable for more complex data types is a question of proper API design that is verbose and scalable but friendly enough to justify the time spent on preparing and using the tool. Trying to write an algorithm automatically handling every possible data type and program state is impractical, hence state comparison, transformation functions and validation data procurement are delegated to the user of an algorithm. By doing this, significantly larger scalability and range of applications can be achieved.

The algorithm then operates on pointers to data without ever needing to cast them back to the original type within the scope of the algorithm execution. Templates are only needed to construct usable function pointers – data manipulation is done on the abstract void pointers or $std :: any$ types. Such a decision is expected to open the tool for extension but close it for modification, allowing it to build algorithms and communication interfaces for it separately, as long as the API contract is honored.

5.1.1. Usage of Aya API from C#

For inter-language communication (e.g. usage of *Aya* from C#), the largest and most challenging issue revolves around creating an API that would make use of *Aya* template based functions.

C# programming language system performs in a managed memory context. This means that every memory allocation, by default, is controlled by a reference-counting garbage collector. If an allocated variable goes out of scope at some point during program execution time, that memory will be freed. It is possible, however, to allocate raw memory using `malloc` wrappers like `Marshal.AllocHGlobal` and to free the allocated memory using `Marshal.FreeHGlobal`. Such strategy allows the use of the provided API almost as if it were a C++ context, albeit it makes the code unsafe, with the portability and reliability of the Common Language Runtime (CLR) being ignored. Moreover, such an approach will inevitably propagate the `unsafe` context to code we might not want to make as such, requiring us to think of ways of containing such code.

```
// Define a function pointer for a testable function
public unsafe delegate void TestableFunction(
    void* a, void* b, out void* c);
[DllImport(...)]
public static extern unsafe void CallTestedFunction(
    TestableFunction func, void* a, void* b, out void* c);
```

Listing 8: C++ Style API usage in C#

CLR provides an extensive API for data marshaling to deal with unsafe code from a managed context. Marshaling is a term that defines data translation between different environments. In this case, it describes a process of data preparation for movement from CLR runtime to a code environment that is native to the underlying platform. For example, a managed interface wrapper for a function mentioned above will look like this:

```
// Define a function pointer for a testable function
public delegate void TestableFunction(
    UIntPtr a, UIntPtr b, out UIntPtr c);
[DllImport(...)]
public static extern void CallTestedFunction(
    TestableFunction func, UIntPtr a, UIntPtr b, out UIntPtr c);
```

Listing 9: Marshaling based API usage in C#

Now, the pointers are converted to the `UIntPtr` type, which stores an address to the memory and can be allocated using the `new` keyword. This allows it to remain within the managed code scope, making such an interface easier and safer to integrate into existing projects.

Major problem related to C++ templates is that pure C++ is not compatible with C# due to name mangling on the C++ side. This means that an external language client must define an intermediate C-like layer that would translate C# pointers into matching C++ data structures and functions. Practically, this means that there should be a *bridge* component written in *cdecl* convention that would call the C# code, and is passed to *Aya* as a testable function.

5.1.2. Final structure of the provided API

Some components of *Aya* are compiled into a dynamic library, which can then be easily integrated into a language of choice by corresponding build systems. Template code is stored in header files that must be provided to the client code.

In the basic scenarios, setting up the interface is straightforward enough to continue work in this area and evolve such a toolset for software testing practitioners who might not be interested in

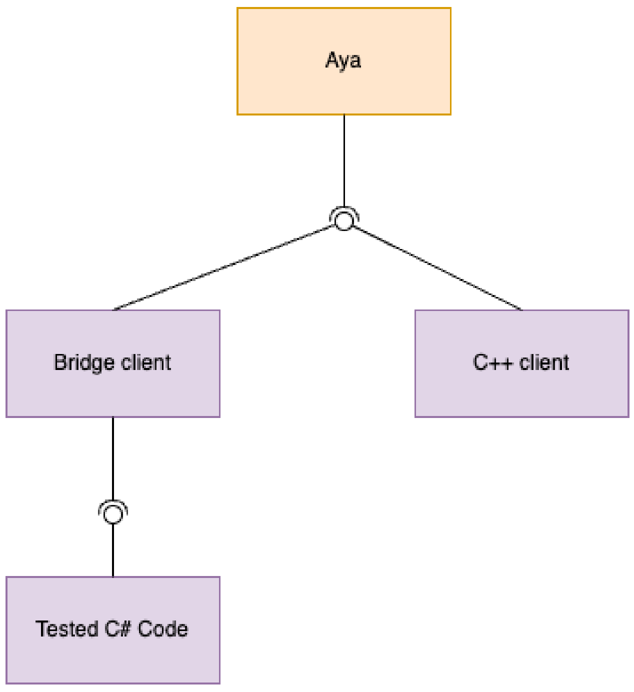
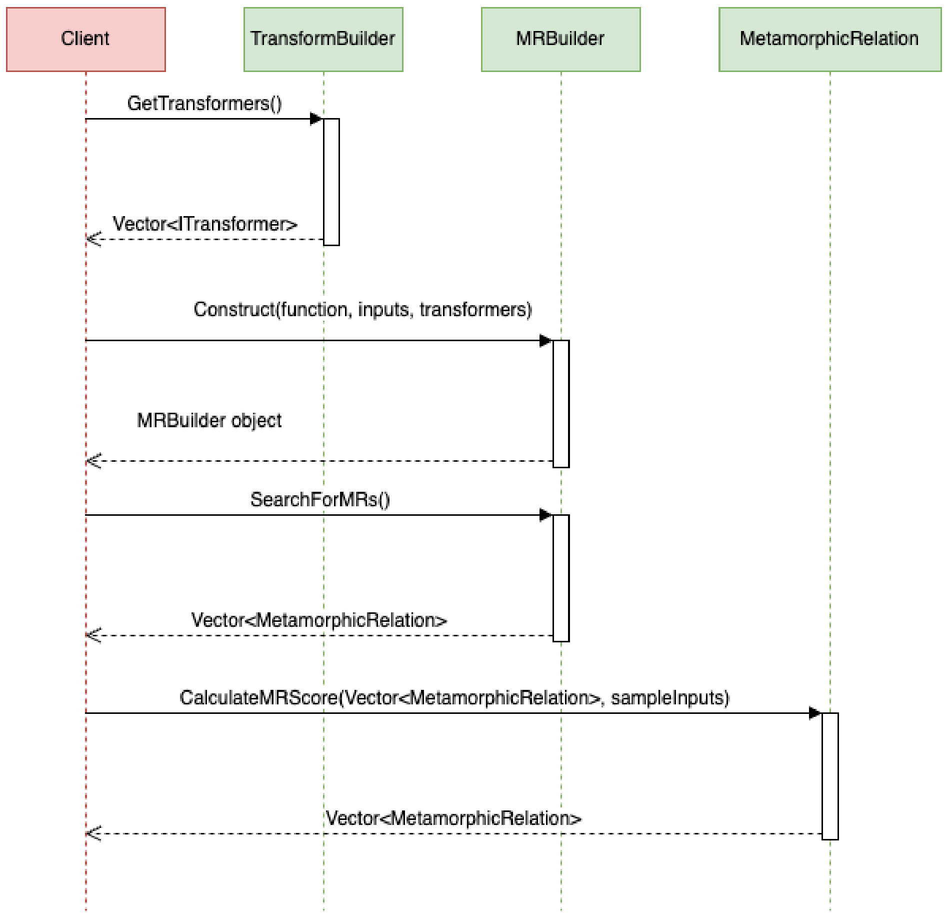


Figure 1. Aya Interface provision. Blocks in purple indicate client code

developing MR generators from scratch. The employed strategy design pattern for the *Aya* API – where we expect a consumer to provide function pointers containing the tested logic and variable transformations, makes such a system more scalable, allowing it to be used across many use cases and programming language systems, compatible with standard *cdecl* naming convention.

A model for a complete procedure of Aya MR search (Fig. 2) indicates two preparation steps – individual transformer construction, and $MRBuilder$ preparation. $MRBuilder$ component generates transform chains of a requested length from given $ITransformer$ instances. Every possible combination is then evaluated during $SearchForMRs()$ call. $CalculateMRScore()$ function is required to check produced metamorphic relations against a different set of inputs. MR score is the percentage of passed MR tests for a given input set. If MRs is valid for 5 inputs out of 10, it's MR score is 50%.



Client
TransformBuilder
MRBuilder
MetamorphicRelation
GetTransformers()
Vector<ITransformer>
Construct(function, inputs, transformers)
MRBuilder object
SearchForMRs()
Vector<MetamorphicRelation>
CalculateMRScore(Vector<MetamorphicRelation>, sampleInputs)
Vector<MetamorphicRelation>

Figure 2. Complete Aya pipeline

5.2. Metamorphic relations as a data structure

In practice, the individual transform chain can be implemented as a dynamic array or a vector and remain efficient enough for the use case. Since the code is written C++, and transform chains are required to be dynamically changed, vectors are the natural choice. Each transformation must be performed in a specific order, always requiring the call of every provided function.

```
struct MetamorphicRelation
{
    vector<ITransformer> inputTransformerChain;
    vector<ITransformer> outputTransformerChain;
}
```

Listing 10: Metamorphic relation data structure

An important addition to the structure of transformers is that certain metamorphic relations require changes to be made with a specific value in the input instead of a constant. Consider Listing 11 as an example. One of the transformers requires the output value to be multiplied by any value stored in i_1. Such scenarios are tested during the MR generation phase, where user-defined input state indices are used to produce temporary transformers.

```
      (TC(Noop(i_1)),
      TC(Sum(i_2, 2), Sub(i_2, 1)),
      OTC(Mul(o, i_1))
```

Listing 11: MR for Pow() with variable transformers

5.2.1. Runtime function signature inference

The following data structure for a metamorphic relation has proven to be useful enough and simple in the described MR generation procedure. The problems may arise when such a data structure needs to be serialized into a file for future reuse due to the inability to effectively serialize function pointers, leaving the main option to store MRs as strings in a file. Mapping a function pointer to a proper function signature mostly depends on the underlying system, in the case of C++, due to specific ABI implementations. While it is possible to use the Runtime Type Identification (RTTI) API to extract the type name in C++, the returned string refers to a mangled name. For demangling, the `abi::__cxa_demangle` function can be used. This function is not guaranteed to work as expected on all platforms since GCC, Clang, or MSVC compilers are implemented differently. Moreover, when the function pointer gets converted to a different type, all function signature information is lost. This means that the function signature must be stored together with a function pointer. Additionally, RTTI can be disabled during program compilation with the `-fno-rtti` flag, killing the possibility of depending on it reliably and adding another reason to control the function metadata separately. This is another observation made during the development of the tool, and it must be addressed in future *Aya* versions.

The situation with runtime function name inference is significantly simpler in the context of C#. It is possible to use the Reflection API, which is guaranteed to work cross-platform. For example, having a delegate for a function, one could call `fptr.GetMethodInfo()` to retrieve all the information needed to form a function call string, which could then be used to produce a serialized test code from constructed metamorphic relations.

Because of currently observed limitations in transform chain serialization, it is required to bundle function pointers as $std :: function$ together with function names as human-readable strings. Such a workaround allows for having MRs stored in text, and if some MRs are proven to be good tests, a user would write a unit test on their own, based on the MR detected. For reference, *Aya* returns MRs which look like this, the one shown in Listing 12.

```
  Mul( Input[0], 1, ) === SinSquared( initialState[1] ) Add(
     initialState[1], -1, ) Div( initialState[1], -1, ) =>
     initialState[0] == followUpState[1]
```

Listing 12: Example of CosSquared() function MR returned by Aya

Such MR could then be converted into a test as shown in listing

```
void test()
{
    double initialInput = 45 * M_PI / 180.0;
    std::vector<double> initialState = {CosSquared(initialInput),
       initialInput};
    double followUpInput = SinSquared(initialState[1]);
    followUpInput -= 1;
    followUpInput *= -1;
    std::vector<double> followUpState = {CosSquared(followUpInput),
       followUpInput};

    ASSERT_EQUAL(initialState[0], followUpState[1]);
}
```

Listing 13: CosSquared test produced from MR

5.2.2. Variable state capture and result equivalency checks

An important problem with a larger variety of searched MRs lies in the fact that it is not obvious how to capture more complex outputs and program states to find MRs. For mathematical functions, where, for sequences of inputs, outputs are another set of scalars, one could use algorithms like cosine similarity check to compare follow-up output and tested output transformation values. In essence, a basic equality operator is enough to compare states. More complex equivalency checks are needed as well for more complex data types, including sequences, to check specific properties of a variable. For that reason, the provision of a comparer function is being delegated to the user because it is assumed that they might be more knowledgeable about the specific value of that property to be checked. For example, if we consider a dictionary data type like `std::map` in C++, we know that if we try adding an element with the same key, the existing value will be overwritten. The `value` property of the map is different, but the `size` remains the same. We could formulate a metamorphic relation stating that for any input with the existing key, the final size of the map will remain unchanged. A user could specify that they are interested in the specific property of a tested structure instead of comparing the total state, which, in the case of custom data structures, may not be valid for comparison. It is also possible to overload the equality $=$ operator for a given structure or a class, and not pass the comparer function. This is a reasonable route if custom code already has overloads for such an operator.

The current algorithm only checks for a state provided by the user without altering the check function in any way, but the following aspect of MR generation is worth investigating. Systematic mutation of a provided comparer function (or iteration over provided checks) would greatly improve the amount of interesting metamorphic relations found. If a state comparison function is described and provided, then it would have to be included in the final MR representation, too, to make the MR reproducible.

Variable state extraction. For runtime type deduction and member location, one could use techniques like C# reflection in runtime and wrap potential fields in comparer function code as delegates, though a similar solution does not exist for C++. Therefore, the most straightforward solution now is to manually provide a collection of possible comparison functions and iterate over them, as we would do in the case of transformers and their corresponding arguments. At the time of writing, the current version of *Aya* does not provide such functionality. This means that to check various states, a user must perform separate MR generation runs.

6. Updated MR search algorithm

The following section contains descriptions of used methods to produce transform chains and generate final metamorphic relations. Practically, every possible combination of input and output transform chains must be checked because any relation might be relevant and valid. Further optimization of created MRs can be considered in the future work.

6.1. Basic MR generation

The algorithm to combine transformations and arguments into MRs was split into several parts: the first step is to generate transform chains that would constitute the final MR. There is an infinite number of possible variants and combinations with which to build MRs, therefore, constraints like a transform chain length are required.

Algorithm 2. Algorithm for Transform Chain generation

```
procedure Transform Chain generation in a search context
    $MaxTransformChainLength$ ←Max length of a TransformChain
    $TransformFunctions$ ←Array of function pointers to transform functions
    $TransformArguments$ ←Array of arrays of pointers to transform function arguments
    $TransformChains$ ←Storage for constructed Transform Chains
    $inputIndex$ ←Index of input array to fill, set to 0
    for each $args \in TransformArguments$ do
        for each $func \in TransformFunctions$ do
            for each $constant \in args$ do
                $StoreTransform(TransformChains[inputIndex], func, constant)$
        $inputIndex \leftarrow inputIndex + 1$
```

Complexity of transform chain generation. Producing transform chains with a certain length is a task that tends to explode in scope pretty quickly. For a $Pow(x, y)$ function, where x and y are both of the same type, having four transformers with at least two arguments each, generating transform chains with length 3 will create $(4 * 2 * 3)^2 = 576$ variants. In practice, the number of iterations greatly depends on the number of sample inputs used to search for MRs. In case of scalar data types like *int*, *double*, etc., it is reasonable to pick the typical smallest and largest possible values alongside some random values. While the complexity of an algorithm is subpar due to it essentially being a brute-force combination check, the area of the search can be limited for the typical use cases.

Algorithm 3. Algorithm for simple MR generation v2

```
procedure MR Generation for program P(X)
    inputArrays ← Randomly generated arrays of inputs
    initialOutputs ← Array of outputs for original inputs
    followUpOutputs ← Array of outputs for transformed inputs
    searchedOutputs ← Array of tested outputs
    InputTransformChainArrays ←Array of TransformChain arrays for inputs
    OutputTransformChains ←Array of TransformChains for output
    createdMRs ←Array of created MRs
    testedFunction ←Pointer to a function being tested
    equalityCheck ←Pointer to an equality check function
    for each $tc1 \in InputTransformChains[0]$ do
        for each $tc2 \in InputTransformChains[1]$ do
            for each $firstInput \in inputs[0]$ do
                for each $secondInput \in inputs[1]$ do
                    $fw1 \leftarrow ApplyTransformChain(tc1, firstInput)$
                    $fw2 \leftarrow ApplyTransformChain(tc2, secondInput)$
                    $ow \leftarrow testedFunction(fw1, fw2)$
                    for each $otc \in OutputTransformChains$ do
                        $sampleOutputs \leftarrow ApplyTransformChain(otc, outputs)$
                        if $equalityCheck(sampleOutputs, ow) == True$ then
                            $ProduceMR(tc1, tc2, otc)$
```

Generating final MRs relies on iterating over the produced transform chains and generated inputs. Input amount can vary, but in practice, it is expected to be the largest amount of information to work with. The expected complexity for this step is quite large: $O(TC^n) * N * O(OTC)$, where TC is the number of transform chains, N is the number of input states, and OTC is the number of output transform chains.

7. MR generation and tests

The following section contains the description of experimentation and testing workflows, performed to validate the Aya MR generator. Conducted experiments aimed to evaluate the performance and validity of MR generation for selected mathematical functions, data structures, and utility functions. The validity of produced MRs is measured by the success rate – a percentage of inputs for which the MR was valid. Success rates help evaluate the general effectiveness of MR search and detect changes in the results after initial setup changes — for example, a check whether the mutation score is another transformation function is used.

Another test was conducted to check how the MR generator reacts to changes in the software under test by employing a mutation testing strategy. The hypothesis is that automated MR generation itself is a test that captures a program's behavior and is reactive to changes in the logic, making such a generator a good candidate for an additional layer of validation.

7.1. Metamorphic relation filtering and validation

Generated MRs, as described, can be executed on an array of inputs as a form of a test. Every MR produced was also converted into such a test during the generation process. Data inputs used in such tests were different from the inputs used to produce such MRs. MR filtering is the next step in the production of an MR list – quite a significant number of MRs are valid for a very small portion of inputs. To account for this, the user can specify the threshold of the success rate.

There were MRs that are valid, but at the same time, meaningless in a context of a test – for example, if transformers are $No - operation$ calls for both initial and the follow-up inputs, a metamorphic relation, essentially, doesn't change any value, and runs the test program with same inputs, comparing same outputs. However, it is not possible to filter out such MRs during the metamorphic relation generation phase without destroying other results. Such filtering is proposed to be done during the post-processing of the final MR list.

7.2. Crash detection

The process of generating metamorphic relations may trigger a crash in a software under test, leading to a potential loss of data. To handle such cases and reduce the external noise, like the likelihood of the crash being caused by the MR generator itself, ideally, the metamorphic relation generator should search for MRs in a controlled environment that would capture crashes. The most common are defects related to memory handling. In managed languages with a Just In Time (JIT) compilation process, every reference is usually checked before being accessed, allowing to throw higher-level exceptions which can be recovered from. This leads to higher safety from memory corruptions at the cost of lower performance [KKN00]. The best possible strategy in lower-level languages like C++ is to capture such crashes using signal handling. In Unix systems, standard handling for a signal like a segfault can be overridden using `sigaction()` system call. Unfortunately, not only is there a limit on the portability of such a handler, but possible usage scenarios of such an override are also limited by the specification of a C++ programming language, especially when handling fatal errors like SIGSEGV. For example, the Listing 14 shows an example of an incorrectly handled crash capture, leading to Undefined Behavior in C++.

```
void crashCapture(int signal)
{
    printf("Crash!");
    some_global_variable++;
}
```

Listing 14: Incorrect signal handling

The signal handler function defined above is incorrect because it causes undefined behavior due to calling an `async-signal-unsafe` function. Effectively, in C standard, if a signal handler does anything other than accessing `sig_atomic_t` data type from within a signal handler, especially calling an I/O operation like `printf()`, the outcome of such operation is considered undefined because a signal could be caught during another I/O call, leading to unpredictable results[Man]. This makes native crash detection risky and impractical. Still, some form of crash prediction is needed, so for now, the best bet is to perform consistent NULL checks in inputs used within tested functions and transformers, and if NULL values are detected, report such MRs as potential crashes.

7.3. Generating MRs for mathematical functions

A simple MR generation evaluation was performed by testing simple mathematical functions from a standard C++ library. Beginning a test by producing MRs for mathematical functions is a natural way of testing MR generation. The majority of such MRs depict known formulas and rules that are valid for such functions – either it's a trigonometric rule like $sin^2 = 1 - cos^2$, or $pow(x, y + 1) = pow(x, y) * x$. The following section is a compilation of results. The first function tested was the power function. Its signature in C++ is:

```
    double pow(double base, double exp);
```

Listing 15: Power function

The transformers used in MR generation were simple arithmetic functions applicable for *double* type, alongside more specific mathematical functions, extracted from the *cmath* library. The following is an example of such MR (Listing 16).

```
        (TC(Noop(i_1)),
        TC(Sum(i_2, 1), Div(i_2, 1), Sub(i_2, 1),
        OTC(Noop(o, i_1))
```

Listing 16: MRs for pow() function

7.3.1. MR generation for reversible computations

Matrix multiplication was tested to test more complex scenarios encompassing bigger data structures instead of simple data types. For testing, two matrix multiplication implementations were used – Apple Accelerate, which implements the BLAS specification, and Unity Mathematics. One of the examples picked was the $2D$ rotation matrix multiplication with a target vector. A rotation matrix has the form:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{13}$$

Combining rotations of various sizes produced metamorphic tests that checked whether a set of rotations that sum up to 360° leaves the vector in the original position within the floating point precision limits. Here is an example of such MRs (Listing 17).

```
    (TC(Rotate120(), Rotate120(), Rotate(120)),
    OTC(Rotate360())
```

Listing 17: MR for rotation matrix

Such metamorphic relations bring up an interesting use case for metamorphic testing, which aims at validating reversible functions – a type of computation that a mirror operation like encryption-decryption procedure can cancel out, or when the operation is reversible to itself, such as the NOT operator. The second variant is very valuable for quantum computing algorithm testing, where each operation is modeled as a unitary matrix – a matrix which, if multiplied by its inverse, returns the identity matrix [YM08].

$$U * U^{-1} = I \tag{14}$$

Testing reversible computing scenarios may be the natural use case for metamorphic testing techniques. Analogous to matrix multiplication, another test conducted was in text encoding-decoding procedures, where a certain text string was converted into a specific encoding, like UTF-8/16/32 and others. The goal was to validate such a conversion library. In the case of C++, *libiconv* was evaluated.

```
    (TC(EncodeString( input[0], UTF-8, )),
    OTC(EncodeString( initialState[0], C99, ) EncodeString(
       initialState[0], UTF-16, ) EncodeString( initialState[0],
       UTF-16, ) EncodeString( initialState[0], C99, ) EncodeString(
       initialState[0], UTF-8, ))
```

Listing 18: Text Encoding MR

As shown in Listing 18, metamorphic relation for encoding/decoding procedures can help testing whether it is possible to make a full ”round-trip” of text encoding without data loss – if both initial and follow-up states, after encoding the text to the same type still produce the same string, no data was lost, and the test has passed.

7.4. Generating MRs for data structures

Another use case of a proposed metamorphic relation generator is the evaluation of data structures. Since a definition for a data structure typically contains a set of functions that can be used to interface with said data structure, combining calls to such methods may represent many usage scenarios, some of which may be proper metamorphic relations. A function for a data structure could be a simple constructor function that returns an initialized and unmodified object or an object with some default starting value. Using member functions as transformers, it is possible to emulate the typical usage scenario, thus making MRs more representative of real-world usage.

```
(TC(Append(base, A) & Append(base, A) & Append(base, A) &
   Append(base, A)),
OTC(Append(output, AA), Append(output, AA)))
```

Listing 19: Most notable MR for std::string with default value constructor

Another tested function implementation was a concatenation of two sequences. With initial values being set to empty strings, identical sets of arguments, and transforms, input and output transform chains were different to accommodate the tested function.

```
(TC(Append(i_1, A)),
TC(Append(i_2, A)),
OTC(Append(output, A), Append(output, A))
```

Listing 20: Most notable MR for std::string with concatenation as a tested function

7.5. Metamorphic testing of Post-Conditions

Metamorphic relation generation for a given function may indicate the existence of certain post-conditions or a contract that is followed by the software under test. Since the generation happens in a black-box scenario, the MR generator ”predicts” the specific change to the input and the output states, which can be later formalized. For example, one could consider a typical ”Tax Calculator” scenario, for which a post condition might look like the following (Listing 21).

```
if income > 60000 then tax = income/3
else if income > 0 then tax = income/4
```

Listing 21: Post-Condition for a tax calculator

If the current implementation of said function is considered correct, the result of MR generation for it can also be considered valid and stored as a test oracle. For future reference, the MR generation process becomes a test, and even the slightest change in generated metamorphic relations will indicate a change in the business logic, which may need to be addressed. While proper unit tests will catch glaring failures, metamorphic tests are meant to capture less obvious changes in the logic in case of more complex conditions or a discrepancy in floating-point computations.

8. Results

In this work, the following results were achieved:

1. Aya, a method of generating MRs for an arbitrary function during runtime, has been proposed.
2. Aya has been tested against typical mathematical functions and more complex data types, such as matrices and strings with encoding metadata, with produced MRs reaching mutation scores on par with analogous solutions.
3. It was observed that the MR generation process is reactive to code mutations, opening up an option of using such a generator as a standalone test.
4. Aya produces MRs by searching for matching input and output transform chains. It was observed that large transform chains are not necessarily more efficient, generating a large number of redundant MRs. Tests suggested that it's best to start with 1-2 transforms in a chain, then increase the size if needed.
5. It was learned that mutation testing allows for efficiently removing redundant MRs, leaving only beneficial metamorphic relations for re-usability in automated tests.

The following topical sections present further discussion and examples of the results.

8.1. Effectiveness of MR Generator

The metamorphic relations were validated additionally by employing mutation testing techniques. For example, the mutations used were a random comparison operator flip, a change to a constant value, a change of a return value, or a random change to one of the arithmetic operators. It was observed that for typical mutations, MRs generated for the original function are losing their previous success rates when being tested against mutated code, indicating a reaction to a mutation. Table 3 shows the average change in non-zero success rates. Mutation testing has also proven to be a valid method of filtering redundant or ineffective MRs – if the metamorphic relation is still valid on a mutated test, then it can be discarded.

Function	Original (%)	Mutated(%)
Tax	96	87
Cos	98	7
Sin	95	5
libiconv.Encode	100	70
Pow	100	80

Table 3. Average change in non 0 % Success Rate MRs

8.2. Effect of Transform Chain Lengths

Conducted measurements of transform chain length efficiency have shown that larger collections of transformations do not necessarily lead to universally better results. The most crucial factor is that specific transformers may cancel out. For example, $mul(x, 2), div(x,2)$ will yield the same outcome as $mul(x,1)$ with a shorter transform chain. For example, every tested transform chain combination could detect mutations in the tax calculator function, effectively reaching a mutation score of 100%. However, the number of redundancies produced was very large — the vast majority of produced MRs had a success rate of 0%. Since many more new redundant combinations are added, the relative efficiency of larger transform chains decreases, as shown in Table 4. For this reason, it may be best to start as small as possible, with transform chains 1-1 or 1-2, and iterate from there. This also coincides with observations made by authors of MR scout, where certain MRs are essentially reversible operations combined, like $push$ and pop methods in a $Stack$ class [XTZ+24].

Input TC Length	Output TC Length	Redundant MRs
1	1	88%
1	2	89%
2	1	90%
2	2	90%
3	1	95%

Table 4. Relation between transform chains and mutation score for the Tax Calculator Function

8.3. Comparison With Analogous Solutions

It is hard to compare Aya to other solutions because most analogous implementations are meant for the Java programming language, while Aya is written in C++. However, similar to GenMorph [ATJ+24], projects tested were standard mathematical functions from the $cmath$ library used in C++. It was decided to compare how MR generators perform to regular unit tests generated for a given function. Tables 5 and 6 show the mutation scores for given functions, achieved by Aya and $GenMorph$, alongside the mutation scores achieved by respective unit tests. Values for GenMorph tests are retrieved from the respective paper [ATJ+24].

Function	Aya MS	LLM-generated unit tests (Grok 3)
sin	100%	100%
acos	90%	100%
tan	80%	100%
log10	100%	100%
pow	57%	100%

Table 5. Achieved Mutation Scores, Aya

Function	GenMorph MS	Generated Unit Tests, Randoop
sin	60%	70%
acos	9%	93%
tan	38%	76%
log10	6%	100%
pow	69%	70%

Table 6. Achieved Mutation Scores, GenMorph, [ATJ+24]

In general, Aya has performed similarly to $GenMorph$, considering the selected transformations and sample inputs used. This observation suggests that the effectiveness of Aya can be on par with the closest analogous MR generation approach.

Conclusions

Having the *Aya* prototype developed and experiments conducted, the following conclusions were made:

1. **MR generator as a test** – Running MR generation during runtime allows capturing program behavior on a real platform. The final distribution of success rates for each MR indicates value ranges for which a particular MR is valid. This observation opens up a potential for the generator to serve as an automated test.
2. **New metamorphic relation patterns** – The Fundamental problem in metamorphic testing is that it heavily depends on how a tester perceives and understands the software they are working with [CKL+18]. This adds bias to metamorphic relations created for the program under test. Automated iteration over potential transformations reduces the effect of bias in MT, thus allowing for new patterns to be discovered since different perspectives are checked.
3. **Reversible computation testing** – Naturally reversible operations like encoding/decoding and matrix operations like vector rotation, by definition, are proper MRs and have been validated as such during this work. This means the metamorphic relation generator becomes a go-to test case generation technique capable of quickly testing reversible functions under various conditions, minimizing the labor needed.
4. **Metamorphic testing as a library** – Universal runtime-based metamorphic relation detection can be integrated into existing projects as a dynamic library, without imposing noticeable changes in the infrastructure needed to start using metamorphic testing techniques. This makes experimentation and spike activities related to metamorphic testing significantly easier, helping to determine whether such a testing method suits the project.
5. **Improvement of a test coverage, not a replacement** – Metamorphic relation can function as a complete test only if it matches the specification of a program. In cases where such information is not present, nor basic test oracles are available to verify the core functionality of a program, metamorphic relations on their own will pose a risk of false security.

References

[AEC22] E. Altamimi, A. Elkawakjy, C. Catal. Metamorphic relation automation: Rationale, challenges, and solution directions. *Journal of Software: Evolution and Process*. 2022, volume 35. Available from: `https://doi.org/10.1002/smr.2509`.

[ASA+20] J. Ayerdi, S. Segura, A. Arrieta, G. Sagardui, M. Arratibel. QoS-aware Metamorphic Testing: An Elevation Case Study. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 2020, pp. 104–114. Available from: `https://doi.org/10.1109/ISSRE5003.2020.00019`.

[ATA+21] J. Ayerdi, V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, M. Arratibel. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In: *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1264–1274. Available from: `https://doi.org/10.1145/3468264.3473920`.

[ATJ+24] J. Ayerdi, V. Terragni, G. Jahangirova, A. Arrieta, P. Tonella. *GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming*. 2024. Available from: `https://doi.org/10.1109/TSE.2024.3407840`.

[BGE+21] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, A. Carzaniga. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software*. 2021, volume 181, p. 111041. issn 0164-1212. Available from: `https://doi.org/https://doi.org/10.1016/j.jss.2021.111041`.

[CKL+18] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, Z. Q. Zhou. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 2018, volume 51, number 1. issn 0360-0300. Available from: `https://doi.org/10.1145/3143561`.

[CR99] S. Cunning, J. Rozenblit. Automatic test case generation from requirements specifications for real-time embedded systems. In: *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*. 1999, volume 5, 784–789 vol.5. Available from: `https://doi.org/10.1109/ICSMC.1999.815651`.

[CT21] T. Y. Chen, T. H. Tse. New visions on metamorphic testing after a quarter of a century of inception. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: Association for Computing Machinery, 2021, pp. 1487–1490. ES-EC/FSE 2021. isbn 9781450385626. Available from: `https://doi.org/10.1145/3468264.3473136`.

[DGR17] A. Di Franco, H. Guo, C. Rubio-González. A comprehensive study of real-world numerical bug characteristics. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 509–519. Available from: `https://doi.org/10.1109/ASE.2017.8115662`.

[Don19] A. F. Donaldson. Metamorphic Testing of Android Graphics Drivers. In: *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. 2019, pp. 1–1. Available from: `https://doi.org/10.1109/MET.2019.00008`.

[FB] FB. *Testing Web Enabled Simulation at Scale Using Metamorphic Testing - Meta Research — research.facebook.com* [`https://research.facebook.com/publications/testing-web-enabled-simulation-at-scale-using-metamorphic-testing/`]. [No date]. [Accessed 03-06-2024].

[GC08] K. Ghani, J. A. Clark. Strengthening Inferred Specifications using Search Based Testing. In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. 2008, pp. 187–194. Available from: `https://doi.org/10.1109/ICSTW.2008.39`.

[Göt23] J. Götborg. *Influence of Automatically Constructed Non-Equivalent Mutants on Predictions of Metamorphic Relations*. 2023.

[HK18] B. Hardin, U. Kanewala. Using Semi-Supervised Learning for Predicting Metamorphic Relations. In: *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*. 2018, pp. 14–17.

[How76] W. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*. 1976, volume SE-2, number 3, pp. 208–215. Available from: `https://doi.org/10.1109/TSE.1976.233816`.

[JCH+16] G. Jahangirova, D. Clark, M. Harman, P. Tonella. Test oracle assessment and improvement. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 247–258. ISSTA 2016. isbn 9781450343909. Available from: `https://doi.org/10.1145/2931037.2931062`.

[jko] jkoritzinsky. *Platform Invoke (P/Invoke) - .NET — learn.microsoft.com* [`https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke`]. [No date]. [Accessed 02-01-2025].

[KB13] U. Kanewala, J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 1–10. Available from: `https://doi.org/10.1109/ISSRE.2013.6698899`.

[KBB15] U. Kanewala, J. Bieman, A. Ben-Hur. Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. *Software Testing, Verification and Reliability*. 2015, volume 26, n/a–n/a. Available from: `https://doi.org/10.1002/stvr.1594`.

[KKN00] M. Kawahito, H. Komatsu, T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *SIGPLAN Not.* 2000, volume 35, number 11, pp. 139–149. issn 0362-1340. Available from: `https://doi.org/10.1145/356989.357002`.

[LAS14] V. Le, M. Afshari, Z. Su. Compiler Validation via Equivalence Modulo Inputs. *ACM SIGPLAN Notices*. 2014, volume 49. isbn 978-1-4503-2784-8. Available from: `https://doi.org/10.1145/2594291.2594334`.

[LLC12] H. Liu, X. Liu, T. Y. Chen. A New Method for Constructing Metamorphic Relations. In: *2012 12th International Conference on Quality Software*. 2012, pp. 59–68. Available from: `https://doi.org/10.1109/QSIC.2012.10`.

[LSN18] X. Lin, M. Simon, N. Niu. Hierarchical Metamorphic Relations for Testing Scientific Software. In: *2018 IEEE/ACM 13th International Workshop on Software Engineering for Science (SE4Science)*. 2018, pp. 1–8.

[Man] L. Manpages. *signal-safety(7) - Linux manual page — man7.org* [`https://man7.org/linux/man-pages/man7/signal-safety.7.html`]. [No date]. [Accessed 09-01-2025].

[McM11] P. McMinn. Search-Based Software Testing: Past, Present and Future. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 153–163. Available from: `https://doi.org/10.1109/ICSTW.2011.100`.

[MKH+08] C. Murphy, G. E. Kaiser, L. Hu, L. Wu. Properties of machine learning applications for use in metamorphic testing. In: *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*. Knowledge Systems Institute Graduate School, 2008, pp. 867–872.

[Nak17] S. Nakajima. Generalized Oracle for Testing Machine Learning Computer Programs. In: *SEFM Workshops*. 2017. Available also from: `https://api.semanticscholar.org/CorpusID:3496745`.

[NME19] A. Nair, K. Meinke, S. Eldh. Leveraging Mutants for Automatic Prediction of Metamorphic Relations using Machine Learning. In: *MaLTeSQuE 2019: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*. 2019, pp. 1–6. Available from: `https://doi.org/10.13140/RG.2.2.30163.94244`.

[RKK20] K. Rahman, I. Kahanda, U. Kanewala. MRpredT: Using Text Mining for Metamorphic Relation Prediction. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 420–424. ICSEW'20. isbn 9781450379632. Available from: `https://doi.org/10.1145/3387940.3392250`.

[SDL+23] C.-A. Sun, H. Dai, H. Liu, T. Y. Chen. Feedback-Directed Metamorphic Testing. *ACM Trans. Softw. Eng. Methodol.* 2023, volume 32, number 1. issn 1049-331X. Available from: `https://doi.org/10.1145/3533314`.

[SDT+17] S. Segura, A. Durán, J. Troya, A. R. Cortés. A Template-Based Approach to Describing Metamorphic Relations. In: *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. 2017, pp. 3–9. Available from: `https://doi.org/10.1109/MET.2017.3`.

[SFP+21] C.-A. Sun, A. Fu, P.-L. Poon, X. Xie, H. Liu, T. Y. Chen. METRIC$^+$+: A Metamorphic Relation Identification Technique Based on Input Plus Output Domains. *IEEE Transactions on Software Engineering*. 2021, volume 47, number 9, pp. 1764–1785. Available from: `https://doi.org/10.1109/TSE.2019.2934848`.

[SG20] H. Spieker, A. Gotlieb. Adaptive metamorphic testing with contextual bandits. *Journal of Systems and Software*. 2020, volume 165, p. 110574. issn 0164-1212. Available from: `https://doi.org/https://doi.org/10.1016/j.jss.2020.110574`.

[SG21] H. Spieker, A. Gotlieb. Summary of: Adaptive Metamorphic Testing with Contextual Bandits. In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2021, pp. 275–277. Available from: `https://doi.org/10.1109/ICST49551.2021.00037`.

[SJW+23] C.-a. Sun, H. Jin, S. Wu, A. Fu, Z. Wang, W. Chan. Identifying metamorphic relations: A data mutation directed approach. *Software: Practice and Experience*. 2023, volume 54. Available from: `https://doi.org/10.1002/spe.3280`.

[SZ09] L. Shan, H. Zhu. Generating Structurally Complex Test Cases By Data Mutation: A Case Study Of Testing An Automated Modelling Tool. *The Computer Journal*. 2009, volume 52, number 5, pp. 571–588. Available from: `https://doi.org/10.1093/comjnl/bxm043`.

[XTZ+24] C. Xu, V. Terragni, H. Zhu, J. Wu, S.-C. Cheung. MR-Scout : Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Transactions on Software Engineering and Methodology*. 2024. Available from: `https://doi.org/10.1145/3656340`.

[XWY19] Z. Xiang, H. Wu, F. Yu. A Genetic Algorithm-Based Approach for Composite Metamorphic Relations Construction. *Information*. 2019, volume 10, p. 392. Available from: `https://doi.org/10.3390/info10120392`.

[YM08] N. S. Yanofsky, M. A. Mannucci. *Quantum Computing for Computer Scientists*. 1st edition. USA: Cambridge University Press, 2008. isbn 0521879965.

[ZCH$^+$14] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, H. Mei. Search-based inference of polynomial metamorphic relations. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 701–712. ASE '14. isbn 9781450330138. Available from: `https://doi.org/10.1145/2642937.2642994`.

[Zhu15] H. Zhu. JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods. In: *2015 Second International Conference on Trustworthy Systems and Their Applications*. 2015, pp. 8–15. Available from: `https://doi.org/10.1109/TSA.2015.13`.

[ZZP$^+$17] P. Zhang, X. Zhou, P. Pelliccione, H. Leung. RBF-MLMR: A Multi-Label Metamorphic Relation Prediction Approach Using RBF Neural Network. *IEEE Access*. 2017, volume 5, pp. 21791–21805. Available from: `https://doi.org/10.1109/ACCESS.2017.2758790`.

Appendices

Source Code for experiments. The prototype written in C++ can be found in this repository: https://github.com/KernalPanik/mt-gen-poc.git

The final source code is in the GitHub environment https://github.com/KernalPanik/Aya. Please refer to the README for the instructions on executing a program.

English-Lithuanian term glossary.

- Metamorphic Relation – Metamorfinis ryšys.Programos savybės, nurodančios, kaip tam tikros įvesčių transformacijos, tam tikroje apibrėžimo srityje keičia išvestis, leidžiančios testuoti programą be tikslaus norimos išvesties žinojimo.
- Metamorphic Testing – Metamorfinis testavimas. Testavimo procesas, paremtas naujų testavimo atvejų kūrimu remiantis metamorfinių sąryšių koncepcija.
- Category-Partition Method – Kategorijų-Dalių metodas. Specifikacija paremtas testavimo atvejų gavimo metodas, paremtas programos skaidymu į būsenų aprašymus vadinamus kategorijomis ir tolimesniu kategorijų padalijimų į rėžius – galimas įvesties ir išvesties reikšmes kategorijoje.
- Search-Based Software Engineering – Paieška grindžiama programų sistemų inžinerija. Programų sistemų inžinerijos atšaka, kurioje problemos sprendžiamos taikant genetinį programavimą bei optimizacijos metodus, iteratyviai pagerinant programos rezultatą naudojant pradinius rezultatus.