Automated and Effective Metamorphic Testing: Relation Discovery, Deduction, and Generation

by

Congying Xu

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfilment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

November 2025, Hong Kong

Automated and Effective Metamorphic Testing: Relation Discovery, Deduction, and Generation

by Congying Xu

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

Abstract

Software testing is a fundamental process for ensuring the reliability and correctness of software systems. Metamorphic Testing (MT), a powerful technique, has been applied in diverse domains to address the oracle problem—a fundamental problem in software testing. However, its broader adoption remains limited due to the difficulty of constructing effective metamorphic relations (MRs), which requires domain-specific expertise. This thesis addresses this challenge by proposing automated approaches for deriving effective MRs.

This thesis makes three main contributions. First, it introduces a novel direction of **discovering and synthesizing MRs from existing tests**. Building on the observation that developer-written test cases often embed domain knowledge that encodes MRs, we propose MR-Scout, an approach that automatically discovers and synthesizes MRs encoded in existing test cases. Using this approach, we discovered over 11,000 MR-encoded test cases from 701 open-source projects.

Second, we focus on **deducing input relations from output relations and examples**. While MR-Scout reveals that thousands of test cases can encode MRs, over 70% of them lack explicit input relations, which hinders their applicability. To address this, we propose MR-Adopt, which leverages large language models to generate input transformation functions that complement these incomplete MRs. Our evaluation shows that MR-Adopt successfully generates valid input transformations for 72% of incomplete MRs.

Finally, we explore **generating MRs directly from a target program**. Although MR-Scout and MR-Adopt effectively derive MRs from existing tests, such MR-encoded tests are relatively few in number, accounting for only 1% of test cases. To overcome this limitation, we propose MR-Coupler, an automated and domain-agnostic approach that generates metamorphic test cases via functional coupling analysis. The key insight is that functional coupling between methods, which is readily available in program code, can be formulated as MRs. This approach successfully generated concrete metamorphic test cases for over 90% of target programs, and the generated test cases successfully revealed 22 real-world bugs.

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Congying Xu

10 November 2025

Automated and Effective Metamorphic Testing: Relation Discovery, Deduction, and Generation

by

Congying Xu

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Prof. Shing-Chi Cheung, Thesis Supervisor

Prof. Xiaofang Zhou, Head of Department

Department of Computer Science and Engineering
10 November 2025

ACKNOWLEDGEMENTS

The acknowledgement is left blank at the stage of thesis proposal.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation and Challenges

Software testing is essential to assure the quality of software systems. **Metamorphic Testing (MT)** has emerged as a powerful testing technique to address a fundamental problem in software testing – the oracle problem [1, 2]. Instead of assessing the outputs of individual inputs, MT works by employing additional test inputs when the expected output for a given input is difficult to determine (i.e., the oracle problem). It reveals a fault if a relation (known as a **Metamorphic Relation (MR)**) between these inputs and their corresponding outputs is violated. Another *outstanding benefit* of MT is that once an MR is identified, MT can leverage a wide range of automatically generated inputs (known as *source inputs*) to exercise diverse program behaviors with no need to prepare oracles for individual inputs [3]. MT has achieved success in detecting faults for various software, such as compilers [4, 5], databases [6, 7], machine translation services [8], and question answering systems [9, 10].

Despite these benefits, the adoption of MT is challenging. A key bottleneck is the construction of effective MRs [1], which requires domain-specific knowledge and relies on the expertise of testers [11]. This hinders the wider adoption of MT [1]. There exist studies trying to systematically explore MRs, such as identifying MRs from software specifications [12, 13] and searching MRs using pre-defined patterns [2, 14] However, these approaches suffer from a low degree of automation, i.e., they heavily rely on manual efforts to identify concrete MRs. On the other hand, several automatic approaches have been proposed to infer MRs for given programs, such as machine-learning-based approaches [15, 16], search-based approaches [17, 18], and genetic-programming-based approaches [19, 20]. However, wide adoption of these approaches is challenging. They are designed for programs in specific domains (e.g., numerical programs [17, 18]) whose input and output values exhibit certain types of relations (e.g., equivalence relations [16], polynomial relations [18], or relations that follow pre-defined patterns [19]).

This thesis aims to address this by proposing automated approaches to derive effective MRs not specific to certain domains or patterns.

1.2 Research Work and Publications

To address the above challenge, this thesis first explores a novel direction of synthesizing MRs from existing test cases. The key *insight* is that domain knowledge encoded in developer-written test cases could suggest useful MRs, even though these test cases may not originally be designed for MT. This thesis proposes MR-Scout, a novel approach to **discover and synthesize MRs from existing test cases**. Based on the experimental findings from MR-Scout, this thesis identifies two key follow-up research problems:

(1) **Incomplete MRs**: 70% of encoded MRs lack explicit input relations, which hinders their applicability to new test inputs for new test generation.

(2) **Sparse MRs**: only 1% of encoded MRs are scattered in only 20% of the studied projects. This hinders the applicability of constructing MRs from existing test cases.

As to the first problem, an LLM-based automated approach (MR-Adopt) is proposed to **deduce input relations from output relations and example input-output pairs**, by generating input transformation functions for MRs encoded in test cases that lack explicit input relations. For the second problem, this thesis explores another novel direction of **generating MRs directly from a target program by analyzing functional coupling**. An LLM-based automated approach (MR-Coupler) is proposed to generate concrete metamorphic test cases directly from source code via functional coupling analysis.

The three approaches increasingly rely on readily available knowledge, advancing the automation of effective metamorphic testing and reducing dependence on expert input.

1.2.1 Synthesis of MRs from existing test cases

With the observation that domain knowledge encoded in developer-written test cases could suggest useful MRs, this thesis refers to such test cases as *MR-encoded test cases* (MTCs) (Figure 3.1). These encoded MRs not only work for existing inputs (e.g., `Text("wow")`) but can be applicable to new inputs (e.g., `Text("wow!")` or `Text("BoldTest")`). This observation motivates us to design an automatic approach to synthesize MRs from existing test cases for automated test case generation.

However, automatically synthesizing MRs that are encoded in test cases presents challenges. On the one hand, there is no syntactic difference between MR-encoded test cases and non-MR-encoded test cases. On the other hand, MRs are implicitly encoded in the test cases. There are no explicit indicators for the detailed constituents (e.g., source and follow-up inputs) of encoded MRs. Discovering these cases is challenging.

In this work, we propose MR-Scout, an automatic approach to discover and synthesize MRs from existing test cases. To tackle the aforementioned challenges, the underlying **insight** of MR-Scout is that MR-encoded test cases actually comply with some properties that can be mechanically recognized. Specifically, MR-Scout works in three phases. MR-Scout first discovers MTCs based on the two derived properties (Section 3.1.1, *MTC Discovery*). Then, with discovered MTCs, MR-Scout deduces the constituents (e.g., source and follow-up inputs) of encoded MRs and then codifies these constituents into parameterized methods to facilitate automated test case generation (Section 3.1.2, *MR Synthesis*). Finally, MR-Scout filters out codified MRs that demonstrate poor quality in applying to new test inputs (Section 3.1.3, *MR Filtering*).

In the evaluation, MR-Scout discovered over 11,000 MTCs from 701 OSS projects. Experimental results show that over 97% of codified MRs are of high quality for automated test case generation, demonstrating the practical applicability of MR-Scout. Furthermore, codified-MRs-based tests effectively enhance the test adequacy of programs with developer-written tests, leading to 13.52% and 9.42% increases in line coverage and mutation score, respectively. Our qualitative study shows that 55.76% to 76.92% of codified MRs are easily comprehensible for developers.

Publication:

- **MR-Scout: Automated synthesis of metamorphic relations from existing test cases.**
  Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, Shing-Chi Cheung
  ACM Transactions on Software Engineering and Methodology, Volume 33, Issue 6, Article 150 (TOSEM 2024)

1.2.2 Deduction of input relations based on output relations and input-output examples

Although the previous work (MR-Scout) reports that developers often encode domain knowledge in test cases that exercise MRs, over 70% of 11,000 MR-encoded test cases (MTCs) in the dataset do not contain explicit input relations. Instead, developers often hard-code the source and follow-up inputs. Without an explicit input transformation program, follow-up inputs cannot be directly generated from automatically generated source inputs. This limitation hinders the reuse of valuable encoded MRs to achieve automated MT and enhance test adequacy.

This work transforms the MR inference problem into a programming by example (PBE) problem. This thesis proposes MR-Adopt (*Automatic Deduction Of inPut Transformation*) to automatically deduce the input transformation from the hard-coded source and follow-up inputs, aiming to enable the encoded MRs to be reused with new source inputs. With typically only one pair of source and follow-up inputs available in an MR-encoded test case as the example, MR-Adopt leverages LLMs to understand the intention of the test case and generate additional examples of source-followup input pairs. This helps to guide the generation of input transformations generalizable to multiple source inputs. Besides, to mitigate the issue that LLMs generate erroneous code, MR-Adopt refines LLM-generated transformations by removing MR-irrelevant code elements with data-flow analysis. Finally, MR-Adopt assesses candidate transformations based on encoded output relations and selects the best transformation as the result.

Evaluation results show that MR-Adopt can generate input transformations applicable to all experimental source inputs for 72.00% of encoded MRs, which is 33.33% more than using vanilla GPT-3.5. By incorporating MR-Adopt-generated input transformations, encoded MR-based test cases can effectively enhance the test adequacy, increasing the line coverage and mutation score by 10.62% and 18.91%, respectively.

Publication:

- **MR-Adopt: Automatic deduction of input transformation function for metamorphic testing.**
  Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Heng cheng Zhu, Jialun Cao
  Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)

1.2.3 Generation of MRs from a target program via functional coupling analysis

Although MR-Scout and MR-Adopt effectively synthesize or deduce MRs from existing tests, such MR-encoded tests are relatively few in number, accounting for only 1% of test cases. Similarly, other existing studies also often rely on knowledge that is hard to obtain. To overcome this limitation, an automatic metamorphic test case generation technique that does not rely on knowledge that is hard to obtain is expected.

This thesis then proposes MR-Coupler, a novel approach that leverages the functional coupling between methods, which is readily available in source code, to automatically construct MRs and generate metamorphic test cases (MTCs). Specifically, MR-Coupler first identifies functionally coupled method pairs, employs large language models to generate candidate MTCs, and validates them through test amplification and mutation analysis. Furthermore, we leverage three functional coupling patterns to avoid expensive enumeration of possible method pairs, and a novel validation mechanism to reduce false alarms. In particular, we leverage three functional coupling patterns to avoid expensive enumeration of possible method pairs, and a novel validation mechanism to reduce false alarms.

Our evaluation of MR-Coupler on 100 human-written MTCs and 50 real-world bugs shows that it generates valid MTCs for over 90% of tasks, improves valid MTC generation by 64.90%, and reduces false alarms by 36.56% compared to baselines. Furthermore, the MTCs generated by MR-Coupler detect 44% of the real bugs. Moreover, the code structures of these MTCs closely follow the human-written MR skeletons. Our results highlight the effectiveness of leveraging functional coupling for automated MR construction and the potential of MR-Coupler to facilitate the adoption of MT in practice. We also released the tool and experimental data to support future research.

Publication:

- **MR-Coupler: Automated Metamorphic Test Generation via Functional Coupling Analysis.**
  Congying Xu, Hengcheng Zhu, Songqiang Chen, Jiarong Wu, Valerio Terragni, Shing-Chi Cheung
  Submitted to the 34th ACM International Conference on the Foundations of Software Engineering (FSE 2026)

1.3 Research Contributions

In summary, this thesis makes the following contributions.

- **Novel directions to derive MRs.** While prior work typically derive metamorphic relations (MRs) from documentation or relying on manual efforts, this thesis explores two novel directions of (1) discovering and synthesizing MRs from existing test cases, and (2) formulating MRs based on functional coupling present in the target program under test.
- **Three automatic and effective approaches to discover, deduce, and generate MRs.** This thesis proposes three approaches to discover, deduce, and generate MRs. The three approaches increasingly rely on readily available knowledge. These approaches promote the automation of effective metamorphic testing and reduce dependence on expert input, thereby lowering the barrier to adoption.

- **Two datasets of numerous developer-written MTCs and real-world bugs detected by MT.** This thesis releases (1) a dataset contains over 11,000 MTCs discovered across 701 OSS projects, and (2) a dataset contains 50 reproducible real-world bugs discovered by MT. These datasets stand as valuable resources for future research in fields such as MR discovery, MR inference, automated MT, and effective MT.
- **Extensive evalutions on proposed approaches.** This thesis conducts extensive experiments to evaluate the effectiveness and pratical usefulness of proposed approaches (MR-Scout, MR-Adopt, and MR-Coupler).

1.4 Thesis Organization

The rest of this thesis is organized as follows. **Chapter 2** provides the background knowledge, including metamorphic testing and the adaptation of MR formulation in the context of OOP (Object-Oriented Programming). **Chapter 3** presents MR-Scout, an automatic approach to discover and synthesize MRs from existing test cases. **Chapter 4** presents MR-Adopt, an LLM-based automated approach to generate input transformations for MRs encoded in test cases that lack explicit input relations. **Chapter 5** describes MR-Coupler, an LLM-based automated approach to generate metamorphic test cases via functional coupling analysis. **Chapter 6** concludes the main research findings and contributions of this thesis, and discusses future research directions.

CHAPTER 2

BACKGROUND AND PRELIMINARIES

2.1 Metamorphic Testing

Metamorphic testing is a process that tests a program ***P*** with a metamorphic relation. Given a sequence of inputs (**source inputs**) and their program outputs (**source outputs**), additional inputs (**follow-up inputs**) are constructed to obtain additional program outputs (**follow-up outputs**). If these inputs and outputs do not satisfy the metamorphic relation, ***P*** contains a fault.

Metamorphic Relation (MR). Let f be a target function. A metamorphic relation of f is a property defined over a sequence of inputs $\langle x_1, \cdots, x_n \rangle$ ($n \geq 2$) and their corresponding outputs $\langle f(x_1), \cdots, f(x_n) \rangle$ [2]. Following the definition by Segura et al. [21], an MR can be formulated as a logical implication from an **input relation** $\mathcal{R}_i$ to an **output relation** $\mathcal{R}_o$.

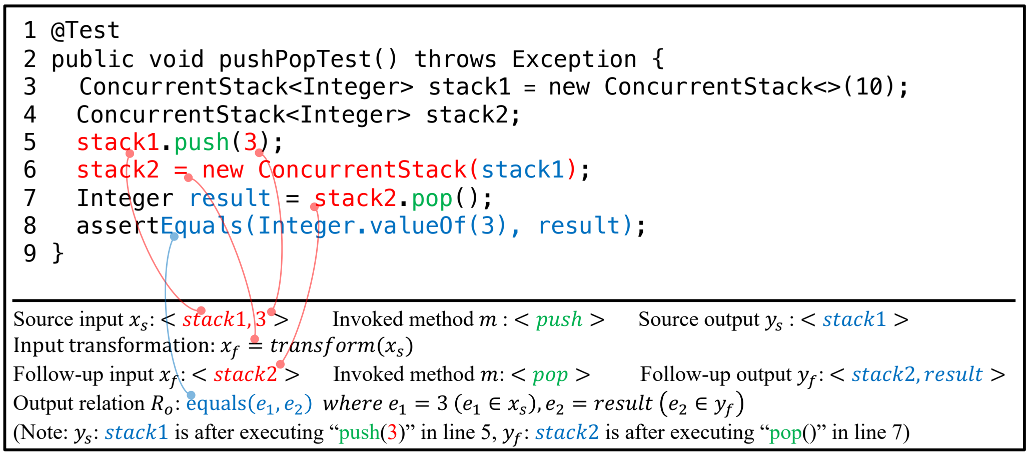
$$\mathcal{R}_i \left(\underset{v=1\cdots k}{\langle x_v \rangle}, \underset{w=(k+1)\cdots n}{\langle x_w \rangle}, \underset{v=1\cdots k}{\langle f(x_v) \rangle} \right) \implies \mathcal{R}_o \left(\underset{i=1\cdots n}{\langle x_i \rangle}, \underset{i=1\cdots n}{\langle f(x_i) \rangle} \right)$$

$\mathcal{R}_i$ is a relation over source inputs $\langle x_1, \cdots, x_k \rangle$, follow-up inputs $\langle x_{k+1}, \cdots, x_n \rangle$, and source outputs $\langle f(x_1), \cdots, f(x_k) \rangle$. The inclusion of source outputs in $\mathcal{R}_i$ allows follow-up inputs to be constructed based on both source inputs and outputs. $\mathcal{R}_o$ is a relation over all inputs $\langle x_1, \cdots, x_n \rangle$ and the corresponding outputs $\langle f(x_1), \cdots, f(x_n) \rangle$. The MR formulation is a general form of that proposed by Chen et al. [22]. It expresses an MR in terms of an input relation and an output relation.

Example 2.1.1. Consider a function $f(a, b, G)$ computing the shortest path from vertex a to vertex b in an undirected graph G. The property $|f(a, b, G)| = |f(b, a, G)|$ implies that the length of the shortest path should be the same in either direction (a to b or b to a), and it can be formulated as

$$x_2 = t(x_1) \implies |f(x_2)| = |f(x_1)| \text{ where } t((a, b, G)) = (b, a, G)$$

In this case, $\mathcal{R}_i = \left\{ \big((v_1, v_2, G), (v_2, v_1, G) \big) \mid \forall G, \forall v_1, v_2 \in G \right\}$ includes all pairs of inputs to f such that the first two elements (source and sink vertexes) are swapped. $\mathcal{R}_o = \{(n, n) \mid \forall n \in \mathbb{N}\}$



```
1 @Test
2 public void pushPopTest() throws Exception {
3   ConcurrentStack<Integer> stack1 = new ConcurrentStack<>(10);
4   ConcurrentStack<Integer> stack2;
5   stack1.push(3);
6   stack2 = new ConcurrentStack(stack1);
7   Integer result = stack2.pop();
8   assertEquals(Integer.valueOf(3), result);
9 }
```

Source input x_s: $< stack1, 3 >$ Invoked method m : $< push >$ Source output y_s : $< stack1 >$

Input transformation: $x_f = transform(x_s)$

Follow-up input x_f: $< stack2 >$ Invoked method m: $< pop >$ Follow-up output y_f: $< stack2, result >$

Output relation R_o: $\text{equals}(e_1, e_2)$ $where\ e_1 = 3\ (e_1 \in x_s), e_2 = result\ (e_2 \in y_f)$

(Note: y_s: $stack1$ is after executing "push(3)" in line 5, y_f: $stack2$ is after executing "pop()" in line 7)

Figure 2.1: A test case crafted from `com.conversantmedia.util.concurrent.ConcurrentStackTest` in project Disruptor. Underlying MR: *x* = *stack.push*(*x*)*.pop*() — *IF* an element *x* is pushed onto a stack and the stack subsequently pops off the top element, *THEN* the element *x* should be the one popped.

includes all pairs of equivalent numbers (shortest paths).

Metamorphic Testing (MT). Given an MR $\mathcal{R}$ for a function f, metamorphic testing is the process of validating $\mathcal{R}$ on an implementation P of f using various inputs [21].

Intuitively, assuming a program implemented by a sequence of statements, MT entails the following five steps [2]: (i) constructing a source input, which can be written by developers or automatically generated (e.g., random testing) [1], (ii) executing the program with the source input to get the source output, (iii) constructing a follow-up input that satisfies R_i, (iv) executing the program with the follow-up input to get the follow-up output, and (v) checking if these inputs and outputs satisfy the output relation $\mathcal{R}_o$.

Input Transformation. In MT, the input relation $\mathcal{R}_i$ is used for constructing the test inputs in the first three steps. Typically, a function, referred to as input transformation, is designed to construct a follow-up input satisfying $\mathcal{R}_i$ from a source input and/or source output. The output relation $\mathcal{R}_o$ serves as the oracle in the last step. For example, in Figure 3.1, the statement `boldTextRder = textRder.text.setBold()` transforms the source input `textRder` to the follow-up input `boldTextRder`, and the output relation `assertTrue(widthNoBold <= widthBold)` gives the oracle.

2.2 Adaptation of MR Formulation in the Context of OOP

Given the observation that developers encode MRs in test cases as oracles (as exemplified in Figure 3.1), our goal is to automatically discover and synthesize these encoded MRs from existing test cases in open-source projects. This work focuses on unit test cases for object-oriented

Listing 1 Illustration of a wrapper function f_c for a stack class implemented with methods `push` and `pop`.
(The output of `fc("push",x)` is a `stack` object which has just pushed `arg` into it, while the output of `fc("pop",x)` are the popped element by executing `stack.pop()` and the `stack` object which has just popped an element.)

```
function fc(m, x) {
// m: unique method identifier
// x: input for executing m, including the receiver object and the arguments
    stack = x.receObj  // receiver object of m
    arg = x.arg        // arguments to m
    switch m:  // fetch the method
        case "push": return stack.push(arg)
        case "pop":  return stack.pop()
}
```

programming (OOP) programs. Since the existing MR formulation is not originally designed for OOP programs, we make a slight adaptation. Specifically, a unit under test refers to a "class" rather than a single function (f) in MR formalism. Therefore, a unit test case for a class under test (CUT) can comprise more than one method invocation. It implies that a metamorphic relation for a class may involve more than one function. For example, in Figure 2.1, the underlying relation $x = stack.push(x).pop()$ is over two functions `push` and `pop` from a stack class.

To accommodate this, we "wrap" the semantics of a class (including its methods) by a function called **class wrapper function** f_c. f_c takes as input a method identifier m and the input x for m, and then invokes $m(x)$ internally. Listing 1 presents an illustration of f_c wrapping a stack class with methods `push` and `pop`. As a result, we can formulate an MR for the stack class based on a single wrapper function instead of functions `push` and `pop`.

Let $f_c(m, x)$ denote the output of f_c invoking the method m on the input x. An MR $\mathcal{R}$ over a sequence of inputs $\langle x_1, \cdots, x_n\rangle$ ($n \geq 2$) with additional corresponding method identifiers $\langle m_1, \cdots, m_n\rangle$ and their corresponding outputs $\langle f_c(m_1, x_1), \cdots, f_c(m_n, x_n)\rangle$ can be formulated as follows.

$$\mathcal{R}_i\Big(\underset{v=1\cdots k}{\langle x_v\rangle}, \underset{w=(k+1)\cdots n}{\langle x_w\rangle}, \underset{v=1\cdots k}{\langle f_c(m_v, x_v)\rangle}\Big) \Longrightarrow \mathcal{R}_o\Big(\underset{i=1\cdots n}{\langle x_i\rangle}, \underset{i=1\cdots n}{\langle f_c(m_i, x_i)\rangle}\Big)$$

For ease of presentation, in the remainder of the work, we use $m(x)$ to denote $f_c(m, x)$, where m is the delegated method in the class under test.

Example 2.2.1. Let f_c stand for the class under test `ConcurrentStack` in Figure 2.1. Given the illustration in Listing 1, the relation $x = stack.push(x).pop()$ can be expressed as: *IF* two inputs $\langle x_1, x_2\rangle$ have the relation $x_2.receObj = push(x_1)$ ($\mathcal{R}_i$), *THEN* the output relation $pop(x_2) = x_1.arg$

($\mathcal{R}_o$) is expected to be satisfied.

In this test case, $x_1.receObj$ and $x_1.arg$ are implemented with `stack1` and `3`, and the invocation $push(x_1)$ is implemented as `stack1.push(3)`. Similarly, $x_2.receObj$ and $pop(x_2)$ are implemented with `stack2` and `stack2.pop()` (`pop()` does not require any argument). The expected relation $pop(x_2) = x_1.arg$ is validated by `assertEquals(Integer.valueOf(3), result)`.

Example 2.2.2. When the function f_c wraps the class `TextRenderer` in Figure 3.1, the relation *IF* $text_2 = text_1.setBold()$ *THEN* $text_1.width() \leq text_2.width()$ can be expressed as: *IF* two inputs $\langle x_1, x_2 \rangle$ have the relation $x_2.receObj\ = x_1.receObj.text.setBold()$ ($\mathcal{R}_i$), *THEN* the output relation $simulateWidth(x_1) \leq simulateWidth(x_2)$ ($\mathcal{R}_o$) is expected to be satisfied.

In this test case, $x_1.receObj$ and $x_2.receObj$ are implemented with `textRder` and `boldTextRder`. Arguments are not needed for `simulateWidth()`, i.e., $x_1.arg = x_2.arg = null$. The statement `assertTrue(widthNoBold <= widthBold)` validates whether the execution results of `textRder.simulateWidth()` and `boldTextRder.simulateWidth()` satisfy the expected output relation $\mathcal{R}_o$.

Note that, for methods with nested method calls, such as $m(g(\cdot))$, the nested method call $g(\cdot)$ is considered as an argument to the method m. $m(g(\cdot))$ can be expressed as $m(x)$ where $x.arg = g(\cdot)$. For methods having side effects, the identification of such methods' outputs will be discussed in Section 3.1.1.

CHAPTER 3

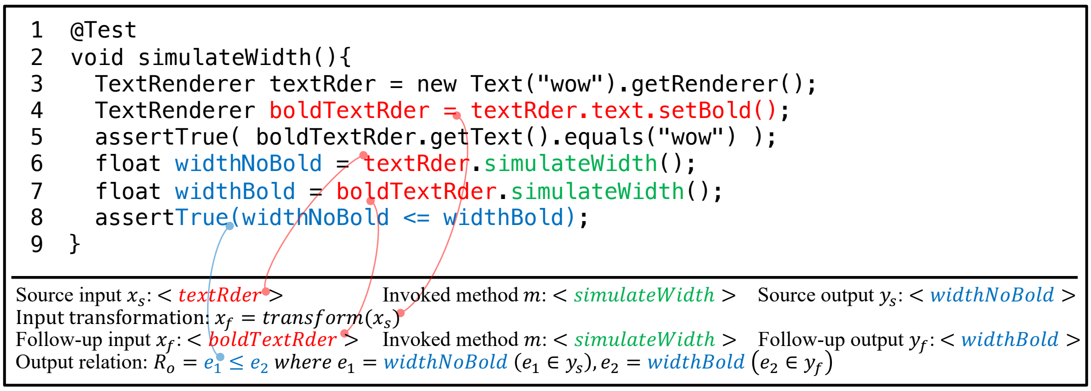
MR-SCOUT: AUTOMATED SYNTHESIS OF RELATIONS FROM EXISTING TESTS

Despite having substantial benefits, the adoption of MT is challenging. A key bottleneck is the construction of effective MRs, which requires domain-specific knowledge and relies on the expertise of testers.

Observation and Idea. The author observes that the domain knowledge encoded in developer-written test cases could suggest useful MRs, even though these test cases may not originally be designed for MT. This thesis refers to such test cases as *MR-encoded test cases* (MTCs). For example, the test case `simulateWidth()` in Figure 3.1 encodes the knowledge that the layout of a text should not be wider than its bold version. This knowledge actually suggests an MR: *IF* $text_2 = text_1.setBold()$ *THEN* $text_1.width() \leq text_2.width()$. Moreover, these encoded MRs not only work for existing inputs (e.g., `Text("wow")`) but can be applicable to new inputs (e.g., `Text("wow!")` or `Text("BoldTest")`). This presents an opportunity of integrating these encoded MRs with automatically generated test inputs to enable automated test case generation [23]. This observation motivates us to design an automatic approach to synthesize MRs from existing test cases for automated test case generation.

However, automatically synthesizing MRs that are encoded in test cases presents challenges. To the best of our knowledge, no existing studies have explored the discovery and synthesis of MRs from existing test cases. On the one hand, there is no syntactic difference between MR-encoded test cases and non-MR-encoded test cases. On the other hand, MRs are implicitly encoded in the test cases. There are no explicit indicators for the detailed constituents (e.g., source and follow-up inputs) of encoded MRs. For the `simulateWidth()` case in Figure 3.1, there is no documentation of the encoded MR in either comments or annotations. After understanding the logic of this test case, the underlying MR and its corresponding constituents can be recognized. This situation presents the challenges of automatically discovering MTCs and deducing the constituents of encoded MRs. Consequently, to discover MRs that are encoded in test cases, our approach needs to analyze whether there is a semantic of MR in a test case.

This work proposes MR-Scout, an automatic approach to discover and synthesize MRs from



```
@Test
void simulateWidth(){
  TextRenderer textRder = new Text("wow").getRenderer();
  TextRenderer boldTextRder = textRder.text.setBold();
  assertTrue( boldTextRder.getText().equals("wow") );
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

Source input x_s: < $textRder$ > Invoked method m: < $simulateWidth$ > Source output y_s: < $widthNoBold$ >
Input transformation: $x_f = transform(x_s)$
Follow-up input x_f: < $boldTextRder$ > Invoked method m: < $simulateWidth$ > Follow-up output y_f: < $widthBold$ >
Output relation: $R_o = e_1 \le e_2$ where $e_1 = widthNoBold$ $(e_1 \in y_s), e_2 = widthBold$ $(e_2 \in y_f)$

Figure 3.1: A test case crafted from `com.itextpdf.layout.renderer.TextRendererTest` in project iText. Underlying MR: *IF* $text_2 = text_1.setBold()$ *THEN* $text_1.width() \le text_2.width()$.

existing test cases. To tackle the aforementioned challenges, the underlying **insight** of MR-Scout is that MR-encoded test cases actually comply with some properties that can be mechanically recognized. Since an MR is defined over at least two inputs and corresponding outputs, this work derives two principal properties that characterize an MR-encoded test case — (i) containing executions of target programs on at least two inputs, and (ii) containing the validation of a relation over these inputs and corresponding outputs.

Specifically, MR-Scout works in three phases. MR-Scout first discovers MTCs based on the two derived properties (Section 3.1.1, *MTC Discovery*). Then, with discovered MTCs, MR-Scout deduces the constituents (e.g., source and follow-up inputs) of encoded MRs and then codifies these constituents into parameterized methods to facilitate automated test case generation. These parameterized methods are termed *codified MRs* (Section 3.1.2, *MR Synthesis*). Finally, MR-Scout filters out codified MRs that demonstrate poor quality in applying to new test inputs. This is because codified MRs that are not applicable to new test inputs are useless for new test generation (Section 3.1.3, *MR Filtering*).

This work built a dataset of over 11,000 MTCs discovered by MR-Scout from 701 OSS projects in the wild. To evaluate the precision of MR-Scout in discovering MTCs, the author and collaborators manually examined 164 randomly selected samples, and found 97% of them are true positives that satisfy the defined properties of an MTC. This indicates the high precision of MR-Scout in discovering MTCs and the high reliability of the MTC dataset (Section 3.2.2, RQ1). MR-Scout synthesizes codified MRs from MTCs and applies filtering to remove low-quality MRs. To evaluate the effectiveness of this process, this work employed EvoSuite to automatically generate a set of new test inputs for each codified MR. Experimental results show that 97.18% of codified MRs are of high quality and applicability to new inputs for automated test case generation, demonstrating the practical applicability of MR-Scout (Section 3.2.3, RQ2). Furthermore, to demonstrate the usefulness of synthesized MRs in comple-

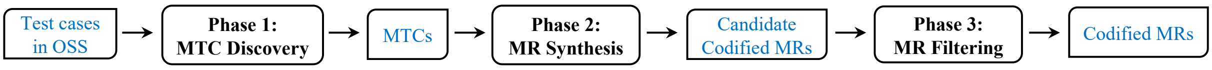


Figure 3.2: Overview of MR-Scout

menting existing tests and enhancing test adequacy, this work compared test suites constructed from codified MRs against developer-written and EvoSuite-generated test suites. Experimental results show 13.52% and 9.42% increases in the line coverage and mutation score, respectively, when the developer-written test suites are augmented with codified-MR-based test suites. As to EvoSuite-generated test suites, there is an 82.8% increase in mutation score (Section 3.2.4, RQ3). To evaluate the comprehensibility of codified MRs, the author conducted a qualitative study involving five participants and 52 samples. Results show that 55.76% to 76.92% of codified MRs are easily comprehended, showcasing their potential for practical adoption by developers.

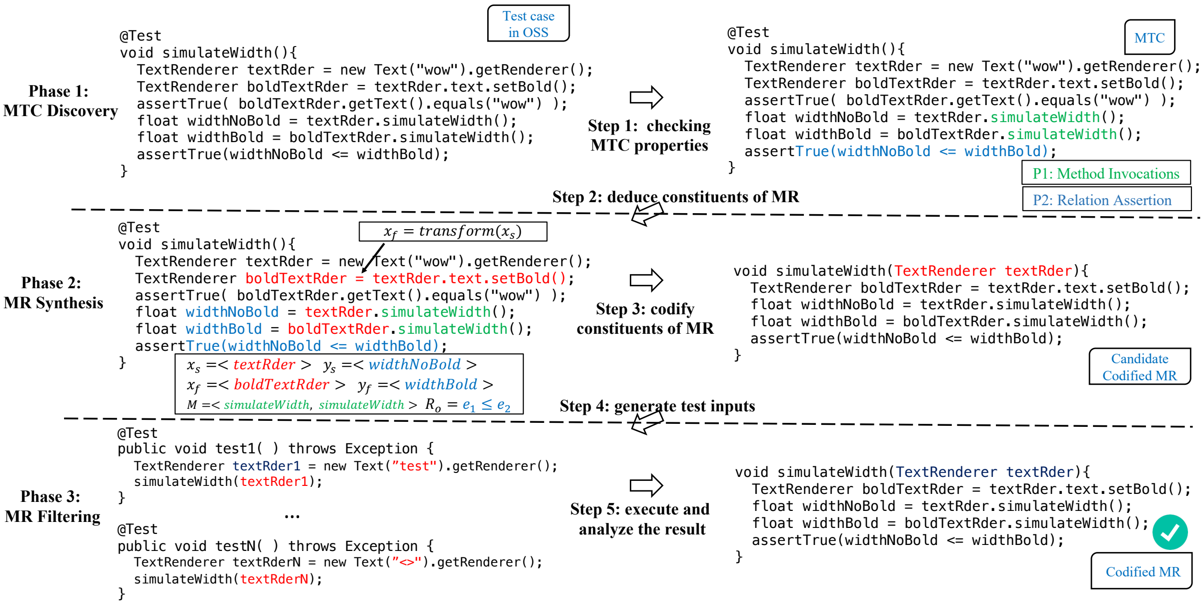
Our work makes the following contributions.

- This work proposes MR-Scout, the first approach that automatically synthesizes MRs from existing test cases.
- This work releases a dataset of over 11,000 MTCs discovered across 701 OSS projects, and investigates their distribution and complexity. This dataset stands as a valuable resource for future research in fields such as MR discovery, MR inference, and automated MT.
- This work conducts extensive experiments to evaluate the precision of MR-Scout in discovering MTCs and evaluate the quality, usefulness, and comprehensibility of MRs synthesized by MR-Scout.
- This work releases the research artifact and all experimental datasets on MR-Scout's website [24] to facilitate reproducing the experimental results and future research.

3.1 Approach

Inspired by the observation that test cases written by developers can embed domain knowledge that encodes MRs, this work proposes an approach, MR-Scout, to discover and synthesize encoded MRs from existing test cases automatically. The underlying insight of MR-Scout is that encoded MRs obey certain semantic properties that can be mechanically recognized. Figure 3.2 presents an overview of MR-Scout. MR-Scout takes as input test cases collected from open-source projects and returns codified MRs. Specifically, MR-Scout works in the following three phases.

(1) *MTC Discovery*. According to the formulation of MR, this work derives two principal



```
@Test
void simulateWidth(){
  TextRenderer textRder = new Text("wow").getRenderer();
  TextRenderer boldTextRder = textRder.text.setBold();
  assertTrue( boldTextRder.getText().equals("wow") );
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

```
@Test
void simulateWidth(){
  TextRenderer textRder = new Text("wow").getRenderer();
  TextRenderer boldTextRder = textRder.text.setBold();
  assertTrue( boldTextRder.getText().equals("wow") );
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

```
@Test
void simulateWidth(){
  TextRenderer textRder = new Text("wow").getRenderer();
  TextRenderer boldTextRder = textRder.text.setBold();
  assertTrue( boldTextRder.getText().equals("wow") );
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

$x_f = transform(x_s)$

$x_s =< textRder > \quad y_s =< widthNoBold >$

$x_f =< boldTextRder > \quad y_f =< widthBold >$

$M =< simulateWidth, simulateWidth > \quad R_o = e_1 \leq e_2$

```
void simulateWidth(TextRenderer textRder){
  TextRenderer boldTextRder = textRder.text.setBold();
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

```
@Test
public void test1( ) throws Exception {
  TextRenderer textRder1 = new Text("test").getRenderer();
  simulateWidth(textRder1);
}
                    ...
@Test
public void testN( ) throws Exception {
  TextRenderer textRderN = new Text("<>").getRenderer();
  simulateWidth(textRderN);
}
```

```
void simulateWidth(TextRenderer textRder){
  TextRenderer boldTextRder = textRder.text.setBold();
  float widthNoBold = textRder.simulateWidth();
  float widthBold = boldTextRder.simulateWidth();
  assertTrue(widthNoBold <= widthBold);
}
```

Figure 3.3: Procedure of MR-Scout operating on the MTC `simulateWidth()`

properties that characterize an MR-encoded test case. First, the test case must contain at least two invocations to methods of the same class with two inputs separately (P1). Second, the test case must contain at least one assertion that validates the relation between the inputs and outputs of the above method invocations (P2). This is because an MR is defined over at least two inputs and corresponding outputs. These two properties guarantee the execution of at least two inputs and the validation of the output relation over these inputs and outputs. By checking the above properties, MR-Scout can mechanically discover MR-encoded test cases (MTCs) in open-source projects (Section 3.1.1).

(2) *MR Synthesis*. Given MR-encoded test cases and corresponding method invocations and relation assertions identified in the *MTC Discovery* phase, MR-Scout first deduces the MR constituents (e.g., source input and follow-up input) and then codifies their constituents into parameterized methods to facilitate automated test case generation. Such methods are termed *codified MRs* in this work (Section 3.1.2).

(3) *MR Filtering*. MR-Scout targets discovering MRs for new test generation. Codified MRs not applicable to new test inputs are ineffective for new test generation [25]. Therefore, in this phase, MR-Scout filters out codified MRs that demonstrate poor quality (e.g., leading to false alarms) in applying to new source inputs.

3.1.1 Phase 1: MTC Discovery

Phase 1 of MR-Scout aims to discover MR-encoded test cases (MTCs). Unfortunately, MTCs are not explicitly labeled and have no syntactic difference with test cases without MRs. There-

Table 3.1: Assertion APIs and examples for relation assertions patterns

Pattern	Assertion APIs in JUnit	Examples

BoolAssert	assertTrue, assertFalse	assertTrue(Math.abs(e_1)>Math.abs(e_2));
		assertTrue(e_1.equalsTo(-e_2));

CompAssert	assertSame, assertNotSame, failNotSame,
	assertEquals, failNotEqual,
	assertArrayEquals, assertThat,
	assertIterableEquals, assertLinesMatch
		assertEquals(e_1, e_2);
		assertEquals(Math.abs(e_2), Math.abs(e_1));

fore, to discover possible MTCs, MR-Scout should analyze whether the given test cases embed the semantics of an MR. So this work first models the semantics of an MR-encoded test case with two principal properties that can be mechanically analyzed. Then, MR-Scout checks these properties in given test cases from open-source projects. Test cases that satisfy the two properties are considered as MTCs by MR-Scout.

Properties of An MTC. According to the formulation of MR in Section 2.1, this work derives two properties (*P1-Method Invocations* and *P2-Relation Assertion*) of an MTC.

P1 Method Invocations: *The test case should contain at least two invocations to the methods of the same class with two inputs separately.* This class is considered as a class under test, and the method invocations are denoted as *MI*. This property is derived from the fact that an MR is defined over at least two inputs and corresponding outputs. When there are at least two method invocations and each method invocation has a pair of input and corresponding output, this ensures the existence of at least two inputs and corresponding outputs. Specifically, the invocations to the same or different methods of a class under test are allowed.

P2 Relation Assertion: *The test case should contain at least one assertion checking the relation between the inputs and outputs of the invocations in MI.* This property is derived from the fact that an MR has a constraint (i.e., $\mathcal{R}_o$) over the input and outputs of program executions (i.e., method invocations). Such an assertion is to validate the output relation $\mathcal{R}_o$.

Step 1: Checking MTC properties. When checking *P1-Method Invocations*, MR-Scout first collects all the method call sites within a test case, and focuses on methods from internal classes that are native to the project under analysis. MR-Scout excludes methods of external classes (such as a class from a third-party library) that are not target classes to test, by matching the prefix of their fully qualified names [26]. For each internal class with at least two method invocations, the class is considered as a class under test. All classes under test and corresponding method invocations are collected to facilitate *P2-Relation Assertion* identification.

However, when it comes to checking *P2-Relation Assertion*, MR-Scout encounters a technical issue: *how to automatically distinguish output relations that are implicitly encoded in assertion statements.* It can be difficult to tell whether an assertion statement represents a genuine relation over multiple outputs or simply a combination of separate output assertions for convenience. For instance, consider an assertion statement with two outputs `y1` and `y2` (e.g., `assertTrue(y1==-1 && y2==1)`). It is ambiguous whether the relation “y1==-y2” should hold or it is a shortcut for `assertTrue(y1==-1)` and `assertTrue(y2==1)`.

To deal with the above issue, this work proposes two general assertion patterns where an output relation can be modeled and validated. Assertions matching these patterns are considered checking an output relation. The design principle of the two patterns is that *an output relation is essentially a boolean expression that relates elements (i.e., inputs and outputs) of method invocations*. This work first introduces the necessary elements of an output relation, and then introduces how these elements should be related.

Necessary Elements of An Output Relation. According to the formulation of MR, an output relation is defined over a set of inputs and outputs. However, there are constraints: (i) the inclusion of a follow-up output, and (ii) the inclusion of either a source output or a source input. As to constraint-(i), the absence of a follow-up output suggests that the second method invocation is not required. This contradicts the definition of MT which requires at least two method invocations. As to constraint-(ii), the absence of a source output and a source input suggests that the first method invocation is not required, contradicting the definition of MT. Note that an output relation can be defined only over a follow-up output and a source output, as illustrated in Figure 3.1 where the follow-up output (`widthBold`) and source output (`widthNoBold`) are included in the output relation. Alternatively, an output relation can be defined only over a follow-up output and a source input. For the case in Figure 2.1, the follow-up output (`result`) and source input (`3`) are included in the output relation.

Given method invocations $MI=\{mi_i\}_{i=1}^{n}$ of a class under test, if an assertion α is verifying an output relation, α must have two elements (donated as e_1 and e_2). e_1 is the input or the output of a method invocation (mi_1), and e_2 is the output of another method invocation (mi_2) that is invoked after mi_1. This allows e_1 to be the source input or output and e_2 to be the follow-up output, satisfying the above two constraints respectively.

Next, this work discusses what are the input x_i and output y_i of a method invocation mi_i. According to the specification of Java [27], method invocation mi_i can be presented as $returnV = receObj.m_i(arg)$, where $returnV$ represents the return value of the invoked method m_i, $receObj$ represents the receiver object of method m_i, and arg represents the input parameter for executing m_i.

- Input x_i is a set, including (i) the input arguments *arg* (primitive values or object references) and (ii) the receiver object *receObj* (if its fields are accessed in the method invocation).
- Output y_i is a set, including (i) the return value *returnV* (if any), (ii) the receiver object *receObj* after the method invocation (if the receiver object's field is updated during the method invocation), and (iii) the objects in *arg* after the method invocation (if these input objects' fields are updated during the method invocation).

Example 3.1.1. For the test case in Figure 2.1, there are two method invocation mi_1 = `stack1.push(3)` on line 5 and mi_2 =`stack2.pop()` on line 7, where $x_1 = \{\texttt{stack1}, \texttt{3}\}$, $x_2 = \{\texttt{stack2}\}$, $y_1 = \{\texttt{stack1}\}$ (just after `stack1.push(3)`), and $y_2 = \{\texttt{result}, \texttt{stack2}\}$ (just after `stack2.pop()`). The assertion α on line 8 can be interpreted as `assertEquals (Integer.valueOf(`e_1`),`e_2`)`, where $e_1 = \texttt{3}$ ($e_1 \in x_1$) and $e_2 = \texttt{result}$ ($e_2 \in y_2$), satisfying the above constraint of elements in an output relation.

Patterns of Relation Assertions. In addition to the above constraint telling if an assertion includes necessary elements (e_1 and e_2) of an output relation, this work further checks if e_1 and e_2 are related by a boolean expression with the following two patterns.

Inspired by existing work on synthesizing assertion oracles with a set of boolean and numerical operators [28], the principle of two patterns is that an output relation assertion should be a boolean expression where necessary elements e_1 and e_2 are related by (i) numerical operators or user-defined boolean methods (*A1-BoolAssert*) or (ii) assertion methods provided by testing frameworks (*A2-CompAssert*).

A1 BoolAssert: For assertions with a boolean parameter, such as `assertTrue`, e_1 and e_2 should be related by (i) numerical operators (i.e., $=, <, >, \leq, \geq, \neq$), or (ii) user-defined methods that return boolean values.

Example 3.1.2. The assertion `assertTrue(widthNoBold <= widthBold)` in Figure 3.1 can be mapped onto *A1-BoolAssert* , where e_1 = `widthNoBold` and e_2 = `widthBold` are related by a numerical operator "$\leq$".

A2 CompAssert: For assertions with parameters for comparison, such as `assertEquals`, one of the parameters should contain e_1, and the other should contain e_2.

Example 3.1.3. The assertion `assertEquals(Integer.valueOf(3), result)` in Figure 2.1 can be mapped onto *A2-CompAssert*, where $e_1 = 3$ and $e_2 = \texttt{result}$. e_1 and e_2 are related by the method `Arrays.equals` which returns a boolean result.

The above two patterns can cover the most commonly used assertion APIs. In Table 3.1, the corresponding APIs in JUnit4 [29] and JUnit5 [30] and some abstract examples of the above two patterns are presented. Assertions that match the two patterns are considered to validate an output relation. It should be noted that there is a trade-off between precision and completeness in recognizing relation assertions. In order to recognize relation assertion precisely, our patterns exclude elements related by logical operators, such as AND, OR, XOR, and EXOR. This is because elements related by these logical operators may not inherently denote a relationship. For example, an assertion `assertTrue(y1 && y2)` can be merely a combination of `assertTrue(y1)` and `assertTrue(y2)` for convenience, without an actual output relation between `y1` and `y2`. While excluding logical operators may cause MR-Scout to miss some output relations, reducing the risk of confusing or misleading developers with incorrect MRs is pretty important.

In a test case, assertions fitting into the above two patterns are considered relation assertions. This indicates that this test case satisfies *P2-Relation Assertion* and is discovered as an MTC. Note that developers may encode more than one MR in a single test case. This work considers the application of an MR for a specific set of inputs and outputs as an **MR instance**. In MTCs, an MR instance is implemented by a relation assertion over the inputs and outputs of method invocations of a class under test. An MR instance in an MTC is denoted as a tuple $\langle □, MI \rangle$, where α denotes a relation assertion and *MI* denotes corresponding method invocations whose output relation is validated by α. MR-Scout collects all MR instances in an MTC to facilitate the following MR synthesis.

Detailed Analysis Process and Limitations. MR-Scout [24] is implemented to statically analyze the source code of test cases. In checking *P1-Method Invocations*, MR-Scout initially collects all method invocations within a given test case. By analyzing the fully qualified names of method invocations, MR-Scout identifies and collects internal classes with at least two method invocations as classes under test. The presence of at least two method invocations in a single internal class indicates that *P1* is satisfied. However, the static analysis of source code may cause imprecise results. For example, the fully qualified names of invoked methods might be wrongly identified if overriding exists. This leads to the satisfaction of *P1-Method Invocations* being wrongly detected.

As to *P2-Relation Assertion*, MR-Scout collects all assertion statements in the given test case. Then, for each assertion, MR-Scout checks whether this assertion is checking the output relation over the inputs and/or outputs of method invocations which are identified in *P1*. During the identification of inputs and output of a method invocation, to tell whether a receiver object is an input and/or an output, MR-Scout analyzes the method's call chain and analyzes whether the fields of the object are accessed or updated in each method of the call chain. However, factors such as aliasing and path sensitivity present challenges in precisely analyzing whether the object's fields are accessed or updated.

3.1.2 Phase 2: MR Synthesis

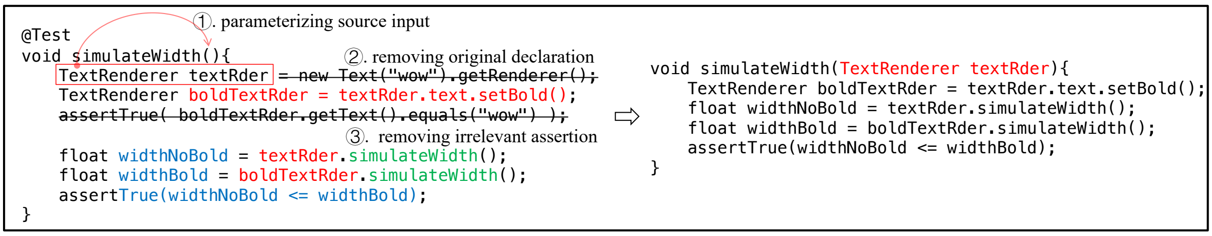
With discovered MTCs in Phase 1, in this phase, MR-Scout synthesizes MRs from these MTCs. However, this process is not straightforward since some encoded MRs are incomplete. Properties *P1-Method Invocations* and *P2-Relation Assertion*, while being principal and necessary, only concern the output relation ($\mathcal{R}_o$) of an MR. Albeit an MR is applied and validated in a test case, the input relation ($\mathcal{R}_i$) can be implicit or even absent. Specifically, for MTCs where the inputs are hard-coded, the input relation is unclear. Inferring the potential relation between hard-coded values is a challenging problem. To the best of the author's knowledge, no existing study explores this problem, nor does similar work. This work focuses on synthesizing MRs from MTCs where input relations are explicitly encoded, i.e., having input transformation that constructs follow-up inputs satisfying $\mathcal{R}_i$ from source inputs and/or source outputs.

The synthesis process involves (i) deducing the constituents of an encoded MR and (ii) codifying these constituents into an executable method that is parameterized with source inputs. This parameterized method is referred to as *codified MR*. By making these methods parameterized with source inputs, new values of source inputs can be easily generated by automatic tools (e.g., Randoop [31] and EvoSuite [32]) and utilized for automated test case generation. These codified MRs are composed of (i) an input transformation, (ii) executions of source and follow-up inputs, and (iii) an output relation assertion.

Step 2: Deducing Constituents of an MR Instance. Developers may encode multiple MRs in a single test case, where a set of MR instances can be identified from an MTC. Following the notations in Phase 1 (Section 3.1.1), for each MR instance $\langle \square, \mathit{MI} \rangle$, MR-Scout deduces a tuple of detailed constituents, including (1) the **target methods**, (2) the **source input**, **follow-up input**, and the **input transformation**, and (3) the **source output**, **follow-up output** and the **output relation assertion**. The details of the deduction are as follows.

(1) MR-Scout takes methods invoked in *MI* as target methods.

(2) As to input-related constituents, MR-Scout first identifies the existence of transformation



```
@Test
void simulateWidth(){
    TextRenderer textRder ~~= new Text("wow").getRenderer();~~
    TextRenderer boldTextRder = textRder.text.setBold();
    ~~assertTrue( boldTextRder.getText().equals("wow") );~~

    float widthNoBold = textRder.simulateWidth();
    float widthBold = boldTextRder.simulateWidth();
    assertTrue(widthNoBold <= widthBold);
}
```

```
void simulateWidth(TextRenderer textRder){
    TextRenderer boldTextRder = textRder.text.setBold();
    float widthNoBold = textRder.simulateWidth();
    float widthBold = boldTextRder.simulateWidth();
    assertTrue(widthNoBold <= widthBold);
}
```

①. parameterizing source input
②. removing original declaration
③. removing irrelevant assertion
⇨

Figure 3.4: Illustration of constructing a codified MR

$x_2 = \mathit{transform}(s)$ ($s \subseteq x_1 \cup y_1$), where s is the input x_1 and/or output y_1 of a method invocation mi_1, and x_2 is the input of mi_2 ($mi_1, mi_2 \in MI$). Then, MR-Scout takes x_1 and x_2 as the source input and the follow-up input, respectively.

Note that not all MR instances have the input transformation because follow-up inputs can be hard-coded rather than constructed from the source inputs and the source outputs. This work only focuses on MRs with input transformation. Besides, MR-Scout synthesizes MRs from MR instances that involve exactly two method invocations ($|MI| = 2$). This is similar to existing studies [1, 2, 12, 14]. Our evaluation results reveal that 64.13% of MR instances only involve two method invocations (Section 3.2.1), indicating that MR-Scout can deal with a large portion of MR instances. Synthesizing MRs from instances that involve more than two method invocations can be challenging and interesting future work.

(3) As to output-related constituents, MR-Scout directly takes the source input corresponding output as the source output, takes the follow-up input corresponding output as the follow-up output, and takes the output relation assertion α in the identified MR instance.

Example 3.1.4. In Phase 2 of Figure 3.3, there is only one MR instance where the output relation assertion α is `assertTrue(widthNoBold <= widthBold)` and the method invocations MI are ⟨`textRder.simulateWidth()`, `boldTextRder.simulateWidth()`⟩.

The identified target method is `simulateWidth()`, the source input is `textRder`, the follow-up input is `boldTextRder`, the input transformation is `boldTextRder = textRder.text.setBold()`, the source output is `widthNoBold`, the follow-up output is `widthBold`, and the output relation assertion α is `assertTrue(widthNoBold <= widthBold)`.

Step 3: Codify Constituents of MR.

This step mainly consists of parameterizing the source input and removing irrelevant assertions. The author illustrates the process of constructing a codified MR using the example shown in Figure 3.4.

An MTC is in the form of a Java method (because a JUnit test case is formatted as a method). To codify MRs as methods parameterized with source inputs, MR-Scout modifies the MTC under codification to take the source input as a parameter. As shown in ① of Figure 3.4, the source input `textRder` is transformed into a parameter to receive new input values. MR-Scout also removes the source input declaration statements (② in Figure 3.4).

Next, MR-Scout removes irrelevant assertions (③ in Figure 3.4). Assertions not identified as relation assertions are considered irrelevant and removed. These irrelevant assertions may be specific to the original the source input and may cause false alarms when new inputs are introduced. In Figure 3.4, assertion `assertTrue( boldTextRder.getText().equals("wow"))` is removed. These modifications enable the codified MR method to receive and validate the relation over values of new source inputs and corresponding outputs.

A codified MR encompasses steps 2-5 of metamorphic testing, involving constructing the follow-up input, executing the target program on both the source input and follow-up input, and validating the output relation across program executions. As a result, automated test case generation can be achieved only when new source inputs are automatically generated to these codified MRs.

3.1.3 Phase 3: MR Filtering

This work aims to discover MRs to generate new test cases, where codified MRs can serve as test oracles. However, codified MRs that are not applicable to new inputs are ineffective for new test generation [25]. Therefore, in this phase, MR-Scout filters out low-quality codified MRs that perform poorly (e.g., leading to false alarms) in applying to new test inputs.

Criterion. Following previous Zhang et al.'s work on inferring polynomial MRs [17], MR-Scout considers *an MR that can apply to at least 95% of valid inputs as a **high-quality MR***. Differently, Zhang et al.'s work infer MRs for numeric programs (e.g., *sin*, *cos*, and *tan*). All generated inputs are numerical values and inherently valid, satisfying the input constraints. In our domain, however, this work is dealing with object-oriented programs whose inputs are not only primitive types but also developer-defined objects. Randomly generated inputs can be invalid. To automatically tell whether an input is valid, the author observes that the program under test contains checks for illegal arguments, and thus assume that *a **valid input** for an MR must be accepted by the input transformation and the methods of the class under test.* That means the execution of a valid test input must not trigger an exception from the statements of input transformation and the invoked methods of the class under test until reaching the relation assertion statement of a codified MR. Note that the checking (not triggering exception) is less stringent than the actual criterion for determining a valid input that satisfies input constraints.

An invalid input might not trigger an exception due to the lack of developer-written checks for illegal arguments and the absence of exception-throwing mechanisms. When an invalid input reaches an assertion statement, it may violate the output relation of a codified MR and produce false alarms. This leads to some high-quality MRs being discarded.

After executing the relation assertion statement, if an `AssertionError` occurs, it indicates the valid input has failed, and the codified MR cannot apply to this input. On the other hand, if no alarm is raised, that means this input complies with the codified MR, thereby the codified MR is applicable to this input.

Inputs Generation. Many techniques have been proposed to generate test inputs, such as random [31], search based [33, 34], and symbolic execution based techniques [35]. Following existing works on test oracle assessment and improvement [28, 36], MR-Scout employs EvoSuite [34] to generate new inputs for codified MRs. Different from Zhang et al.'s work where MRs are for *sin*, *cos*, and *tan* programs, and 1000 new numeric inputs can be easily generated for each MR, in our domain, the inputs of codified MRs are not only primitive types but also developer-defined objects. For MRs with complex objects as inputs, EvoSuite cannot generate a large amount (e.g., 1000) of valid objects as new inputs. So this work gives the same time budget rather than the same amount of inputs for each codified MR. In line with the configuration of previous works [37, 38], for each codified MR, MR-Scout runs EvoSuite 10 times with different seeds and gives a time budget of 2 minutes for each run. The detailed configuration of EvoSuite can be found on MR-Scout's website [24].

Then, MR-Scout executes these test cases (as illustrated in Figure 3.3 (Phase 3)) and analyzes the execution result (i.e., pass or fail). Finally, MR-Scout outputs high-quality codified MRs that can apply to at least 95% of generated valid inputs.

3.2 Evaluation

Our evaluation aims to answer the following research questions:

RQ1 Precision: Are MTCs discovered by MR-Scout possessing the derived properties of an MTC? (Section 3.2.2)

RQ2 Quality: How is the quality of MR-Scout codified MRs in applying to new inputs for automated test case generation? (Section 3.2.3)

RQ3 Usefulness: How useful are MR-Scout codified MRs in enhancing test adequacy? (Section 3.2.4)

RQ4 Comprehensibility: Are MRs codified by MR-Scout comprehensible? (Section 3.2.5)

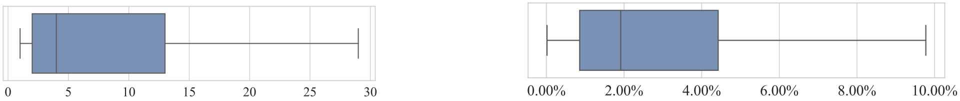
RQ1 aims to evaluate the precision of MR-Scout in discovering MTCs, i.e., whether discovered test cases possess the defined properties of an MTC. To answer RQ1, this work first ran MR-Scout, and 11,350 MTCs from 701 projects were discovered. Then, the author and collaborators manually analyzed 164 sampled MTCs. **RQ2** aims to evaluate the quality of MR-Scout codified MRs, by using a set of new test inputs not present in the filtering phase of MR-Scout. The results also indicate the effectiveness of the *MR Filtering* phase in the methodology. **RQ3** is to evaluate the usefulness of codified MR when integrated with automatically generated inputs. Specifically, this work analyzes whether test suites constructed from codified MRs can enhance test adequacy on top of developer-written and EvoSuite-generated test suites. **RQ4** aims to assess whether MR-Scout codified MRs are easily comprehensible for developers engaged in tasks like test maintenance or migration. For this purpose, the author conducted a small-scale qualitative study on 52 codified MRs.

3.2.1 Dataset Preparation

This work selected open-source projects from GitHub [39]. This work chose public projects meeting these criteria: (i) labeled as a Java project, (ii) having at least 200 stars, and (iii) created after 01-January-2015. These criteria enable us to analyze high-quality and contemporary Java projects that are more likely to use mature unit testing frameworks like JUnit [40] and TestNG [41]. The number of stars indicates the popularity and correlates with project quality [42]. This work considered projects created after 01-January-2015 to exclude old projects that might require obsolete dependencies and frameworks, while some popular Java projects that were created before 2015 might be excluded.

By 05-April-2022, 7,395 projects met these criteria and were collected for experiments. These projects account for 71.49% of all popular Java projects that have at least 200 stars and were created both before and after 01-January-2015. This work cloned the latest version of each selected project at that time and collected tests from these projects. This work considered methods annotated with "`@Test`" as tests and files containing tests as test files. This work excluded 3,327 projects without tests. At last, this evaluation had 4,068 projects, which contained 1,021,129 Java tests in 545,886 test files. These projects comprised 239,724,897 lines of production code and 80,130,804 lines of test code.

MTC Discovery. This work ran MR-Scout on each of the 4,068 projects on a machine with dual Intel® Xeon™ E5-2683 v4 CPUs and 256 GB system memory. For the *MTC Discovery* phase, MR-Scout took 18 hours and 58 minutes, with an average analysis time of 16.78 seconds



(a) Number of MTCs in a project (b) Percentage of MTCs in a project

Figure 3.5: Distribution of 11,350 MR-encoded test cases (MTCs) in 701 projects w.r.t the number and percentage

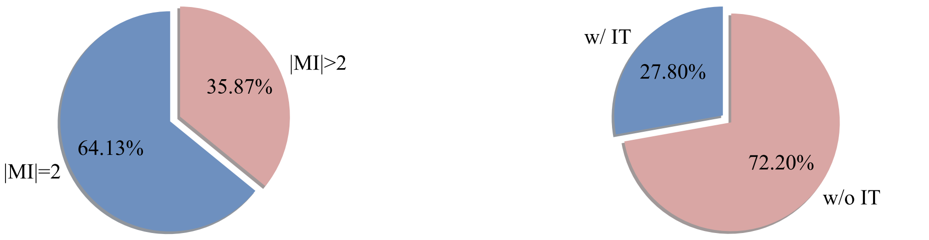
per project. Finally, 11,350 MTCs in 701 (17.23%) projects were discovered. On average, each project has 16.19 MTCs.

As to 3,367 (82.77%) projects where no MTC was discovered, our observations suggest a limited presence of test cases encoded with MRs. Typically, test cases in these projects are structured to assert whether the actual output of a method under test aligns with the expected output for a given input. Moreover, some projects exhibited inadequate testing, having few test cases. This reduces the chances of discovering MTCs. Additionally, MR-Scout is designed to discover MRs of a class. This means that MR-Scout focuses on MRs associated with methods within a single class. MRs over multiple classes or at higher levels are out of the scope of MR-Scout.

Distribution of MTCs. The distribution of MTCs provides insights into how MTCs are spread across projects. The distribution of 11,350 MTCs in the 701 projects varies significantly, ranging from a single MTC to 500 MTCs. As shown in Figure 3.5a, the majority of the projects have 1 to 29 MTCs, and the median is 4. Half of the 701 projects have 2 to 13 MTCs. This work further analyzed the percentage of MTCs among all tests in each project in Figure 3.5. For the majority of projects, 0.02% to 9.78% of the tests are MTCs, and the median is 1.91%. Percentages of MTCs in half of these projects range from 0.8% to 4.42%.

This work also examined the top 25 projects with the highest number of MTCs (the projects can be found at [24]). These projects span various domains, including complex data structures, data processing, distributed computing, data visualization, smart contracts, website building, code parsing, and more. The results indicate that MTCs are broadly distributed across projects from diverse domains rather than being concentrated within a few projects with specific functionalities.

Complexity of MTC. The discovered 11,350 MTCs contain a total of 21,574 MR instances (introduced in Section 3.1.2). On average, 1.90 MR instances were found per MTC. 13,836 (64.13%) out of the 21,574 MR instances involved only two method invocations. Among these 13,836 MR instances, 3,847 (27.80%) instances in 2,743 MTCs were associated with an input transformation. This indicates that a significant proportion (64.13%) of MR instances leverage MRs involving only two method invocations, and 72.20% of MR instances are without input



(a) Size of involved MI (|MI|)

(b) Existence of IT, when |MI|=2

Figure 3.6: Distribution of 21,574 MR instances w.r.t the size of involved method invocations (|MI|) and the existence of an input transformation (IT)

transformation. In this work, this work targets synthesizing MRs from MR instances involving two method invocations and having input transformation.

These results indicate that numerous MR-encoded test cases are widely spread across open-source projects of different domains. 17.23% of projects contain MTCs, and in total 11,350 MTCs were discovered from 701 projects. Besides, the majority of encoded MR instances (64.13%) involve relations with two method invocations. The MTC dataset is released and available on MR-Scout's site [24].

MR Synthesis and Filtering. This work focuses on MTCs where MR instances (i) involve two method invocations ($|MI| = 2$), (ii) have input transformation, and (iii) are from compilable projects where mutation analysis and EvoSuite-based MR filtering can be conducted. Finally, in MR-Scout discovered MTCs, this work collected 485 MTCs from 104 projects.

In *MR Synthesis* phase, 441 (90.92%) MTCs' encoded MRs were successfully synthesized into compilable codified MRs. The other 9.08% of MTCs failed due to too complicated external dependencies or code structures.

In *MR Filtering* phase, MR-Scout filtered candidate codified MRs with inputs generated by EvoSuite. EvoSuite could successfully generate valid inputs for 125 candidate codified MRs. Among 125 candidate codified MRs that have valid inputs, 60.00% (75/125) passed the *MR Filtering* phase and were finally outputted by MR-Scout as high-quality codified MRs. The main reasons why some codified MRs were not generated with valid inputs include too complex preconditions, incompatible environment, violation of input constraint, etc., which are detailly discussed in Section 3.3.

3.2.2 RQ1: Precision

Experiment Setup. MR-Scout discovers MR-encoded test cases based on static analysis of the source code. However, factors such as aliasing, path sensitivity, dynamic language features like reflection, and handling of recursions can cause imprecise analysis results. Therefore, in RQ1, this work aims to evaluate whether MR-Scout is precise in discovering MR-encoded test cases in real-world projects. The results reflect the reliability of our released dataset of discovered MTCs.

To achieve this goal, the author and collaborators manually validate if the MTCs discovered by MR-Scout have the two properties mentioned in Section 3.1.1. However, there are 11,350 MTCs discovered in our evaluation subjects, and it is infeasible to check all of them manually. Therefore, this work randomly sampled 164 out of 11,350 MTCs to estimate the precision of MR-Scout. Such a sample size can be calculated by an online calculator [1], ensuring a confidence level of 99% and a confidence interval of 10% for our estimation result [43].

During the validation, two participants (PhD students in my research group) independently inspected the source code of the discovered test cases. Based on their understanding, they labeled a test case with one of the following labels:

- *True Positive* indicates a test case possesses the two properties mentioned in Section 3.1.1.
- *False Positive* indicates a test case does not possess the two properties.
- *Unclear* indicates the participants cannot understand a test case or tell if the two properties are possessed.

After independent labeling, the two participants discussed test cases labeled differently or labeled as *Unclear* and finally reached a consensus.

Result. During the manual validation, there were 27 test cases assigned with different labels by the two participants, and the divergences were resolved. Finally, among the 164 sampled test cases, 160 cases were labeled as true positives, whereas the remaining four were labeled as false positives. Overall, based on a sampled dataset, MR-Scout demonstrates an estimated precision of 97% (with a confidence level of 99% and a margin of error at 10%) in discovering MTCs.

All of the four false positives were due to incorrect identification of *P2-Relation Assertion*. This is because MR-Scout does not well handle the scopes of variables in complicated cases with re-assigned variables and non-sequential control flows. Listing 2 shows a simplified example, where `m` and `n` are assigned with the return values of method `CUT.abs` in the class under test. When encountering assertion `AssertEquals(m,n)`, MR-Scout mistakenly considers this assertion to fulfill *P2-Relation Assertion* — validating the relation over outputs of `CUT.abs(x)` and `CUT.abs(x*x)`. Consequently, MR-Scout falsely considers this test case to be positive. However, before assertion, the variables `m` and `n` may be re-assigned with the return values of `min(x,x*x)` and `max(x,x*x)` which are not methods in the class under test. `AssertE-`

[1] https://www.qualtrics.com/experience-management/research/determine-sample-size/

Listing 2 Simplified example of a false positive MTC

```
    ...
        m = CUT.abs(x);
        n = CUT.abs(x*x);
    ...
        if(m>n){
            m = Math.min(x,x*x);
            n = Math.max(x,x*x);
        }
        assertTure(m <= n); // false positive output relation assertion
    ...
```

quals(m,n) is a false positive output relation assertion.

Despite a minor fraction of false positives, most discovered MTCs satisfy the properties mentioned in Section 3.1.1. The results show that MR-Scout can effectively discover MTCs in real-world projects, and our dataset is of high reliability.

Answer to RQ1: MR-Scout is precise in discovering MTCs in real-world projects, achieving an estimated precision of 97% in discovering MTCs. Such high precision indicates the high reliability of our released dataset of MTCs.

3.2.3 RQ2: Quality

Experiment Setup. In *MR Filtering* phase, MR-Scout filters out low-quality MRs with new inputs generated by EvoSuite within a certain time budget. This approach differs from Zhang et al.'s work [17], which infers MRs for numeric programs MRs and generates 1,000 numeric inputs for each MR. The reason for our choice is the difficulty of generating a vast quantity of complex objects as inputs for some MRs in our domain. This choice potentially weakens the *MR Filtering* phase in the methodology. Therefore, in this RQ, this work aims to evaluate the quality of MR-Scout codified MRs in applying to new inputs for automated test case generation. The result also indicates the effectiveness of the *MR Filtering* phase.

To achieve this goal, this work reused the criterion of **a high-quality MR** defined in Section 3.1.3. MRs that are not of high quality are termed low-quality MRs. Besides, considering the *MR Filtering* phase has already relied on EvoSuite-generated inputs, we re-ran EvoSuite with different seeds and had a replication check to construct a set of different test inputs for evaluation, thereby mitigating the circularity issue in the evaluation.

In this RQ, this work evaluates the quality of 75 codified MRs that are output by MR-Scout after filtering (Section 3.2.1). In line with the configuration of the previous studies [37, 38], this work re-ran EvoSuite 10 times with different seeds to mitigate the randomness issue on the

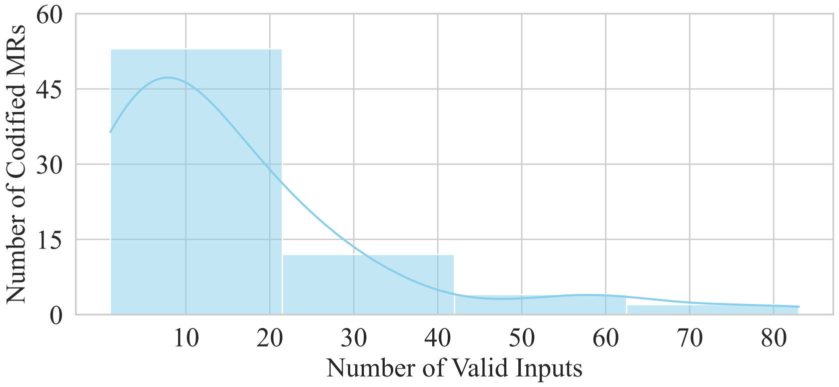


Figure 3.7: Distribution of generated valid inputs

evaluation results and gave a time budget of 2 minutes for each run[2]. With the generated inputs, this work filtered out inputs that appeared in the *MR Filtering* phase of MR-Scout, and filtered out invalid inputs according to the criterion of a valid input in Section 3.1.3. Finally, 4 codified MRs did not have newly generated valid inputs. 71 codified MRs had 1,995 generated inputs, where 57.69% (1,151) of them are valid inputs, with an average of 16.21 valid inputs for each codified MR. Figure 3.7 shows the distribution of generated test inputs.

Result. Out of 71 MR-Scout output MRs, the author found that 97.18% (69) of MRs are high-quality and even applicable to all valid inputs. Two codified MRs are low-quality. 16 (out of 24) valid inputs of the two codified MRs result in `AssertionError` alarms. After manually analyzing, the author found that the 2 codified MRs are indeed of low quality. For example, the simplified MR $width(text) < width(text.setBold())$ asserts that the layout of bold text should be wider than non-bold text. However, this MR cannot apply when a text is empty or contains only characters that cannot be bold (e.g., "<>") or the original text is already bold.

Answer to RQ2: The *MR Filtering* phase in our methodology is effective. 97.18% of MR-Scout synthesized MRs are of high quality and applicability to new inputs for automated test case generation, demonstrating the practical applicability of MR-Scout.

3.2.4 RQ3: Usefulness

Experiment Setup. This RQ aims to evaluate the practical application of MRs synthesized by MR-Scout when combined with automatically generated test inputs. Specifically, one application scenario of synthesized MRs is testing the original programs where these MRs are found, and this work focuses on assessing how useful codified-MRs-based tests are in complementing existing tests and improving the test adequacy of these original programs.

[2] The detailed configuration of EvoSuite can be found on MR-Scout's website [24].

Table 3.2: Enhancement of test adequacy by codified-MR-based test suites ($\mathcal{C}$) on top of developer-written ($\mathcal{D}$) and EvoSuite-generated test suites ($\mathcal{E}$)

Metrics	**VS.** $\mathcal{D}$			**VS.** $\mathcal{E}$			**VS.** $\mathcal{D}$+$\mathcal{E}$		
	$\mathcal{D}$	$\mathcal{D}$+$\mathcal{C}$	**Enhancement**	$\mathcal{E}$	$\mathcal{E}$+$\mathcal{C}$	**Enhancement**	$\mathcal{D}$+$\mathcal{E}$	$\mathcal{D}$+$\mathcal{E}$+$\mathcal{C}$	**Enhance.**
Line Coverage	0.5769	0.6549	+13.52%	0.3735	0.5682	+52.10%	0.6351	0.6785	+6.83%
Test Strength	0.7162	0.7366	+2.86%	0.2420	0.4889	+102.03%	0.6977	0.7369	+5.62%
% of Covered Mutants	0.5960	0.6757	+13.37%	0.3675	0.5389	+46.63%	0.6598	0.7057	+6.95%
Mutation Score	0.5032	0.5506	+9.42%	0.1789	0.3271	+82.80%	0.5395	0.5823	+7.93%

Metrics and Baselines. This work employs the following four metrics to measure the test adequacy.

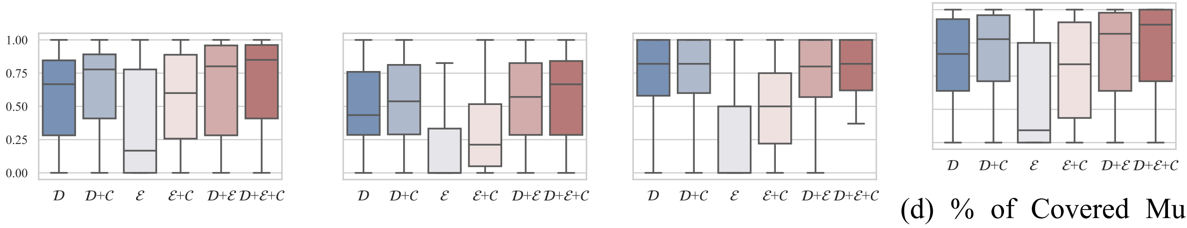
- Line Coverage: the percentage of target programs' lines executed by a test suite.
- Test Strength: the percentage of executed mutants killed by a test suite.
- Percentage (%) of Covered Mutants: the percentage of mutants executed by a test suite.
- Mutation Score: the percentage of mutants killed by a test suite.

Firstly, codified MRs are integrated with automatically generated valid inputs in the RQ2 (Quality) to construct codified-MR-based test suites (denoted as $\mathcal{C}$). Then, this work compares the performance of the codified-MR-based test suite on these four metrics against two baselines: (i) developer-written test suites ($\mathcal{D}$) and (ii) EvoSuite-generated test suites ($\mathcal{E}$). Note that both the developer-written test suite and the EvoSuite-generated test suite target all methods in the class under test, while a codified MR only invokes MR-involved methods, which is a subset of all methods in the class under test. Thus, this work does not directly compare the performance of the codified-MR-based test suite against developer-written or EvoSuite-generated test suites. Instead, this work investigates whether the codified-MR-based test suites can enhance the test adequacy on top of developer-written and EvoSuite-generated test suites.

The author successfully ran PIT [44], a mutation testing tool, to generate 2,170 mutants for 51 target classes of 75 codified MRs (which were collected in the dataset preparation, Section 3.2.1). There are a total of 4,701 lines of code in these target classes.

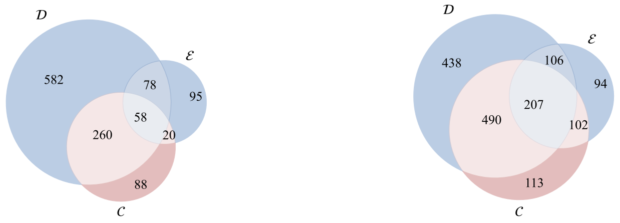
Statistical Analysis. This work performed a statistical analysis (i.e., Mann-Whitney U-test [45, 46]) to test the hypothesis — the fault detection capability of test suites augmented with codified MR-based tests (i.e., $\mathcal{C}$+$\mathcal{D}$+$\mathcal{E}$) is better than existing tests (i.e., $\mathcal{D}$+$\mathcal{E}$). Specifically, this work compares the fault detection capability based on killed mutants. For each mutant, if it is killed by a test suite, the score for this mutant is 1, otherwise 0. Finally, for the Mann-Whitney U-test, test suites $\mathcal{C}$+$\mathcal{D}$+$\mathcal{E}$ and $\mathcal{D}$+$\mathcal{E}$ will get a list of scores for all 2170 mutants, respectively.

Result. Table 3.2 presents the results of the four metrics on 51 classes. Compared with $\mathcal{D}$,



(a) Line Coverage (b) Mutation Score (c) Test Strength (d) % of Covered Mutants

Figure 3.8: Enhancement of test adequacy by codified-MR-based test suites ($\mathcal{C}$) on top of developer-written ($\mathcal{D}$) and EvoSuite-generated test suites ($\mathcal{E}$)



(a) Killed mutants (b) Covered mutants

Figure 3.9: Comparison of covered and killed mutants by developer-written ($\mathcal{D}$), EvoSuite-generated ($\mathcal{E}$), and codified-MR-based ($\mathcal{C}$) test suites

incorporating $\mathcal{C}$ leads to a 13.52% increase in the line coverage, and 13.37% and 9.42% increases in the percentage of covered mutants and mutation score. Compared with $\mathcal{E}$, incorporating $\mathcal{C}$ leads to a remarkable 82.8% increase in mutation score and 52.10% in line coverage. Even compared with the test suites combining $\mathcal{D}$ and $\mathcal{E}$ (i.e., $\mathcal{D}$+$\mathcal{E}$), incorporating $\mathcal{C}$ can still achieve 6.83% and 7.93% enhancement in line coverage and mutation score. The result indicates that test suites constructed from MR-Scout discovered MRs can effectively improve the line coverage and mutation score, showing the fault-revealing capability of test suites.

Figure 3.8 presents box-and-whisker plots showing the comparison results of test suites ($\mathcal{C}$, $\mathcal{D}$, and $\mathcal{E}$) on the four metrics. We can find that no matter compared with $\mathcal{D}$ or $\mathcal{E}$ or $\mathcal{D}$+$\mathcal{E}$ test suites, incorporating $\mathcal{C}$ leads to an overall enhancement in terms of the median, first quartile, third quartile, upper and lower whiskers (1.5 times IQR) of four metrics.

Figure 3.9 illustrates the numbers of mutants covered and killed by developer-written ($\mathcal{D}$), EvoSuite-generated ($\mathcal{E}$), and codified-MR-based ($\mathcal{C}$) test suites. Codified-MR-based test suites cover 215 (+11.37%) more mutants and kill 108 (+9.42%) more mutants, compared with existing developer-written test suites. Even compared with the combination of developer-written and EvoSuite-generated test suites, codified-MR-based test suites have 113 (+6.95%) exclusively covered mutants and 88 (+7.93%) exclusively killed mutants. Furthermore, the result of the *t*-test shows that our hypothesis is retained. The fault detection capability of test suites augmented with codified MR-based tests (i.e., $\mathcal{C}$+$\mathcal{D}$+$\mathcal{E}$) is significantly better than existing tests (i.e., $\mathcal{D}$+$\mathcal{E}$) (p-value=0.003 < 0.05 which is a typical threshold of significance). The corresponding effect size (i.e., normalized U statistic) is 0.52. These results indicate the usefulness of codified MRs in enhancing the test adequacy (i.e., the test coverage and fault-detection capability).

The enhanced test adequacy by codified-MR-based test suites results from the effective integration of high-quality test oracles (i.e., codified MRs) with a set of diverse test inputs. In developer-written test suites, although test oracles are well-crafted and invaluable, each oracle typically applies to one test input. EvoSuite-generated test suites have a large number of test inputs but fall short in the quality of test oracles [32, 47].

Codified-MR-based tests merge the merits of both developer-written tests and EvoSuite-generated tests. When compared with developer-written tests, codified-MR-based tests leverage the same reliable test oracles but with a greater diversity of random test inputs that explore more branches of the target programs. When compared with EvoSuite-generated tests, codified-MR-based tests do not have a higher quantity of test inputs, but offer rich developer-crafted test oracles and more meaningful sequences of method invocations (since codified MRs are structured by at least two method invocations, i.e., *P1-Method Invocations* in Section 3.1.1). EvoSuite was designed to generate only five types of assertions [32]. Nevertheless, EvoSuite's random generation of test inputs and sequences cannot succeed in invoking some methods that require complex pre-conditions. The corresponding examples and detailed analysis can be found in MR-Scout's website [24]. As a result, codified-MR-based test suites can effectively improve both line coverage and mutation score.

Answer to RQ3: Test cases constructed from codified MRs lead to `13.52%` and `9.42%` increases in line coverage and mutation score for programs with developer-written test suites, demonstrating the practical usefulness of codified MR in complementing existing tests and enhancing test adequacy.

3.2.5 RQ4: Comprehensibility

Experiment Setup. This work considers that MR-Scout synthesized MRs are useful not only for testing their original programs but also for testing other programs that share similar functionalities. In such usage scenarios, when an MR is easy to understand, it simplifies the debugging and maintaining processes. Furthermore, comprehensible MRs facilitate test migration for other programs with similar functionalities. Therefore, this work designs RQ4 to assess the comprehensibility of codified MRs synthesized by MR-Scout.

To this end, this work conducted a small-scale qualitative study with five PhD participants who are experienced in programming in Java and MT. Specifically, all participants have more than one year of experience researching MT-related topics, and more than three years of programming in Java and using JUnit.

Procedure. To reduce manual efforts, the author randomly sampled 52 cases from the 75 MR-Scout synthesized codified MRs (which were collected in the dataset preparation, Section 3.2.1) for the qualitative study. Such a sample size can be calculated by an online calculator [3], ensuring a confidence level of 99% and a confidence interval of 10% for our analysis result [43].

For each codified MR, the participants were required to understand (i) the logic of the MR and (ii) the relevance of this MR to the class under test. Then, the participants rate the comprehensibility of this MR. To avoid neutral answers, participants express their opinions using a 4-point Likert scale [48] (i.e., 1: very difficult to understand, 2: difficult to understand, 3: easy to understand, and 4: very easy to understand).

Statistical Analysis. After participants rated the comprehensibility of sampled MRs, we performed a statistical analysis (i.e., one-sample t-test [45, 46]) on the rating results. The one-sample t-test is a statistical method to test hypotheses about whether the mean of one group of samples differs from a given value.

The author first aggregated the ratings for each codified MR. Specifically, for each codified MR, the author calculated the average of the comprehensibility scores given by the raters (denoted as $\bar{X}$). Then, the author tested the hypothesis — the mean of $\bar{X}$ over the sampled MRs is greater than 2.5, where 2.5 represents a neutral score.

Result. Figure 3.10 shows the participants' responses to the comprehensibility of codified MRs. Overall, 55.76% to 76.92% of the sampled codified MRs are easy (or very easy) for participants to understand. Moreover, 15.38% to 34.61% of codified MRs are scored as very easy. However, there are still 23.08% to 44.24% of the sampled codified MRs that are difficult (or very difficult) to understand. The result of the one-sample t-test shows that our hypothesis is retained. Specifically, the mean comprehensibility of sampled MRs is significantly greater than the neutral score μ=2.5 (p-value=$3.46\times10^{-6} < 0.05$ which is a typical threshold of significance), and the corresponding effect size (i.e., Cohen's d) is 0.70. These results indicate that codified MRs are comprehensible.

The author also gathered feedback from participants to investigate the factors that make synthesized MRs difficult to understand. The author found that the main difficulty in understanding some MRs is from the complexity of certain classes under test. The test cases in the evaluation were collected from highly-starred Java projects, which often exhibit complex structural depen-

[3] https://www.qualtrics.com/experience-management/research/determine-sample-size/

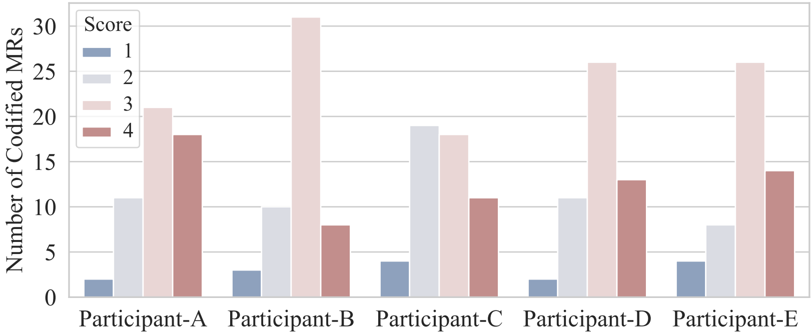


Figure 3.10: Comprehensibiliy scores of 52 MR-Scout synthesized MRs (Score: 1. very difficult, 2. difficult, 3. easy 4. every easy to understand)

dencies between classes (Section 3.2.1). In the qualitative study, participants were required to understand the relevance between an encoded MR and the class under test. Some classes are too complicated for participants to understand their functionalities and business logic, thus making it difficult to understand the relevance. However, it is important to note that, for developers who actively maintain these projects or seek to migrate these test oracles (i.e., codified MRs) to similar functionalities in other programs, the codified MRs might be relatively simpler to understand. Familiarity with the projects would likely mitigate the difficulties posed by class complexity.

Answer to RQ4: 55.76% to 76.92% of codified MRs can be easily comprehended, showcasing the potential of codified MRs for practical adoption by developers engaged in test maintenance and migration.

3.3 Discussion

3.3.1 Threats to Validity

The author has identified potential threats to the validity of our experiments and has taken measures to mitigate them.

Subjectivity in Human Judgment. The evaluation of precision (RQ1) and comprehensibility (RQ4) depends on human judgment. To reduce potential subjectivity and misjudgments, the author gave the participants a training session before manual validation. for RQ1, two authors independently validated samples, and then collaboratively resolved any uncertainties or disagreements and came to a consensus, ensuring a rigorous cross-checking mechanism. For RQ4, participants without sufficient experience in MT, Java, and JUnit may affect the results. To mitigate the threats, all involved participants had a solid background in MT, Java, and JUnit, establishing a consistent level of expertise as a baseline for evaluation.

Sampling Bias. The evaluation of precision (RQ1) and comprehensibility (RQ4) is based on randomly sampled cases. Different samples may result in different results. To mitigate this threat, our sample size statistically ensures a confidence level of 99% and a confidence interval of 10% for our evaluation result.

Representativeness of Experiment Subjects. A possible threat is whether our findings on the selected OSS projects can be generalized to other popular projects. To mitigate this threat, the author first adopted criteria from previous empirical studies on OSS projects [49, 50] to select high-quality and well-maintained Java projects, as described in Section 3.2.1. Then, the author quantified the coverage of our selected projects to all popular Java projects on GitHub. The result shows that the selected projects account for 71.49% of all popular projects that have at least 200 stars and were created before the cut-off date of our evaluation (i.e., 05-April-2022). This coverage suggests that the selected projects are representative.

EvoSuite Configuration. The choice of parameters for EvoSuite, such as search budget, time limit, and seeds, might affect valid inputs generated by EvoSuite. When EvoSuite generates different valid inputs for evaluation, the results of the quality (RQ2) and usefulness (RQ3) of codified MRs can be different. To mitigate this threat, the author followed the practices of existing studies [37, 38] to run EvoSuite 10 times and chose appropriate parameters that fit our scenario.

3.3.2 Applications of MR-Scout

Metamorphic testing is an approach to both test result verification (i.e., test oracle problem) and test case generation [2] based on metamorphic relations (MRs). MR-Scout aims to synthesize MRs from existing test cases that encode domain knowledge and suggest useful MRs. Such MRs are useful for testing not only their original programs but also other programs that share similar functionalities.

As to testing original programs, MR-Scout synthesized MRs help test case generation. Synthesized MRs are in the form of parameterized methods, which can be easily integrated with automated input generators to enable automated test case generation. This results in a higher fault-detection capability (as evaluated in Section 3.2.4: Usefulness). Furthermore, codified MRs, representing properties of target programs, help describe the behaviors of classes under test across potential test inputs, simplifying test maintenance.

3.3.3 Limitations and Future Work

Despite MR-Scout being effective in discovering and synthesizing high-quality and useful MRs from existing test cases, MR-Scout still has several limitations.

1. MR-Scout only considers MR instances that involve exactly two method invocations because the constituents of their encoded MRs are unambiguous and easier to identify than MR instances involving more than two method invocations. Synthesizing MRs from instances that involve more than two method invocations can be challenging and interesting future work.
2. MR-Scout only considers MTCs with explicit input relation (i.e., input transformations). Synthesizing MRs from MTCs without explicit input relations could be challenging and interesting future work.
3. MR-Scout statically analyzes the source code of test cases. Factors such as aliasing, path sensitivity, and dynamic language features can cause imprecise analysis results, as discussed in Section 3.2.2. Our sampling result (4 false positives out of 164 samples) reveals that this problem is relatively minor in practice.
4. MR-Scout synthesized MRs can be of low quality and cause false alarms. To discard such low-quality MRs, this work designed a filtering phase based on the pass ratio (i.e., at least 95%) of valid inputs. However, MR-Scout determines the validity of an input based on developer-written checks (such as `IllegalArgumentException` statements). When such checks are lacking, invalid test inputs may reach assertion statements, violate the output relation, and produce false alarms. Due to the lack of checks for invalid test inputs, MR-Scout cannot differentiate between false alarms and true bug-exposing alarms. The filtering phase in MR-Scout may discard some high-quality and bug-exposing MRs. Effectively distinguishing the validity of an input and assessing the quality of MRs could be interesting future work.
5. MR-Scout employs EvoSuite-generated inputs to evaluate the quality and usefulness of codified MRs. However, EvoSuite is coverage-based and ineffective in generating a large number of valid inputs. Here are several main reasons [51]: (i) Complex precondition of codified MRs: the time budget may not be enough for EvoSuite to construct complex objects that involve many dependencies or deep hierarchies; (ii) Incompatible environment: EvoSuite can be incompatible with some libraries or dependencies in the target project; (iii) Bugs of EvoSuite: EvoSuite has bugs that cause crashes during generating inputs for codified MRs; (iv) Violation of input constraint: some EvoSuite-generated inputs did not

conform to the expected input format or constraints (e.g., strings meeting the “mm/dd/yy” date format); (v) Invalid call sequence: the precondition for invoking a method is not satisfied (e.g., the requirement of invoking `setup()` first is not satisfied in the EvoSuite-generated test sequence). As noted in [51], “other prototypes are likely to suffer from the same problems we face with EvoSuite.” Generating complex objects in the real world remains a challenge for automatic tools.

3.4 Related Work

3.4.1 MR Identification

Many studies proposed MRs for testing programs of various domains (e.g., compilers [52–55], quantum computing [56], and AI systems [8, 57–61]). This work reviews and discusses the most closely related work in systematically identifying MR.

MR Pattern Based Approaches. Segura et al. [23] proposed six MR output patterns for Web APIs, and a methodology for users to identify MRs. Similarly, Zhou et al. [14] proposed two MR input patterns for testers to derive concrete metamorphic relations. These approaches simplify the manual identification of MRs but have limitations: (i) MR patterns are designed for certain programs (such as RESTful web API), (ii) requiring manual effort to identify concrete MRs, and (iii) MR patterns only cover certain types of relations (e.g., equivalence) are not general to complicated or customized relations. In contrast, MR-Scout automatically discovers and synthesizes codified MRs without manual effort and is not limited to MRs of certain programs or certain types. Chen et al. proposed METRIC [12], enabling testers to identify MRs from given software specifications using the category-choice framework. METRIC focuses on the information of the input domain. Sun et al. proposed METRIC+ [13], an enhanced technique leveraging the output domain information and reducing the search space of complete test frames. Differently, MR-Scout synthesizes MRs from test cases and does not require the software specification and test frames generated by the category-choice framework.

MR Composition Based Approaches. The MR composition techniques were proposed to generate new MRs from existing MRs. Qiu et al. [25] conducted a theoretical and empirical analysis to identify the characteristics of component MRs making composite MRs have at least the same fault detection capability. They also derive a convenient, but effective guideline for MR composition. Different from these works, MR-Scout does not require existing MRs and can complement these approaches by providing synthesized MRs for composition-based approaches.

Search and Optimization Based Approaches. Zhang et al. [17] proposed a search-based approach for automatic inference of equality polynomial MRs. By representing these MRs with a set of parameters, they transformed the inference problem into a search for optimal parameter values. Through dynamic analysis of multiple program executions, they employed particle swarm optimization to effectively solve the search problem. Building upon this, Zhang et al. [18] proposed AutoMR, capable of inferring both equality and inequality MRs. Firstly, they proposed a new general parameterization of arbitrary polynomial MRs. Then, they adopt particle swarm optimization to search for suitable parameters for the MRs. Finally, with the help of matrix SVD and constraint-solving techniques, they cleanse the MRs by removing the redundancy. These approaches focus on polynomial MRs, while MR-Scout considers MRs as boolean expressions, allowing for greater generalization.

Ayerdi et al. [19] proposed ayerdi2021generating, a genetic-programming-based approach to generate MRs automatically by minimizing false positives, false negatives, and the size of the generated MRs. However, MRs generated by ayerdi2021generating are limited to three predefined MR Input Patterns. Sun et al. [62] proposed a semi-automated Data-Mutation-directed approach, μMT, to generate MRs for numeric programs. μMT makes use of manually selected data mutation operators to construct input relations, and uses the defined mapping rules associated with each mutation operator to construct output relations. However, MRs generated by μMT are limited to pre-defined mapping rules. In comparison, MR-Scout has no such constraints, applicable for more than numeric programs.

Machine Learning Based Approaches. Kanewala and Bieman [15] proposed an ML-based method that begins with generating a control flow graph (CFG) from a function's source code, extracts features from the CFGs, and then builds a predictive model to classify whether a function exhibits a specific metamorphic relation. Building upon this, Kanewala et al. [63] further identified the most predictive features and developed an efficient method for measuring similarity between programs represented as graphs to explicitly extract features. Blasi et al. [16] introduced MeMo, which automatically derives metamorphic equivalence relations from Javadoc, and translates derived MR into oracles. Different from Memo, MR-Scout is not limited to equivalence MRs. These approaches rely on source code or documentation to discover MRs. MR-Scout complements these approaches by synthesizing MRs from test cases.

3.4.2 Parameterized Unit Tests

Parameterized unit tests (PUTs) are tests that accept parameters. A single PUT can be executed with varying input values. PUTs offer several advantages in software testing. PUTs are applied with a range of test inputs that can be automatically (e.g., using EvoSuite [32]) to exercise paths of the methods under test. The high test coverage typically results in a better fault-detection capability compared to conventional unit tests. Unlike conventional unit tests, PUTs can take parameters that can be bound to a set of values, allowing exploration of more program states by a single test, making maintenance easier, and reducing test redundancy.

Several studies have been conducted to generate PUTs. Fraser et al. [64] proposed to generate PUTs from scratch using a genetic algorithm to generate method-call sequences and using mutation analysis to construct test oracles. Kampmann et al. [65] assumed the existence of high-quality system tests and proposed to automatically extract parameterized unit tests from system test executions. Thummalapenta et al. [66] proposed a methodology (termed Test-Generation) to help developers retrofit conventional unit tests into PUTs.

MR-Scout differs from these earlier studies by synthesizing the underlying metamorphic relations from existing unit tests. Additional unit tests can be automatically generated based on the synthesized relations. The methodology of MR-Scout is orthogonal to those adopted by these studies, which have different assumptions and application scenarios. Furthermore, Thummalapenta et al. [66] aimed to study the costs and benefits of converting unit test cases into parameterized unit tests. The work conducted an empirical study and proposed a methodology to help developers manually promote inputs as parameters, define test oracles, add assumptions, and construct mock objects based on existing test cases. In contrast, MR-Scout automatically synthesizes codified MRs from existing test cases.

3.5 Chapter Conclusion

Developers embed domain knowledge in test cases. Such domain knowledge can suggest useful MRs as test oracles, which can be integrated with automatically generated inputs to enable automated test case generation. Inspired by the observation, this work introduce MR-Scout to automatically discover and synthesize MRs from existing test cases in OSS projects. This work models the semantics of MRs using a set of properties. MR-Scout first discovers MR-encoded test cases based on these properties, and then synthesizes the encoded MRs by codifying them into parameterized methods to facilitate new test case generation. Finally, MR-Scout filters out low-quality MRs that demonstrate poor quality in their applicability to new inputs for automated test case generation.

MR-Scout discovered over 11,000 MR-encoded test cases from 701 OSS projects. Experimental results show that MR-Scout achieves a precision of 0.97 in discovering MTCs. 97.18% of the MRs codified by MR-Scout from these test cases are of high quality and applicability for automated test case generation, demonstrating the practical applicability of MR-Scout. Moreover, test cases constructed from these synthesized MRs can effectively improve the test coverage of the original test suites in the OSS projects and those generated by EvoSuite, demonstrating the practical usefulness of MR-Scout synthesized MRs. Our qualitative study shows that 55.76% to 76.92% of the MRs codified by MR-Scout can be easily comprehended, showcasing the potential of synthesized MRs for practical adoption by developers.

Data Availability. We make MR-Scout and the experimental data publicly available at MR-Scout's site [24] to facilitate the reproduction of our study and relevant studies of other researchers in the community.

CHAPTER 4

MR-ADOPT: AUTOMATIC DEDUCTION OF INPUT TRANSFORMATION FUNCTION

One *outstanding benefit* of Metamorphic Testing (MT) is that once an Metamorphic Relation (MR) is identified, MT can leverage a wide range of automatically generated inputs (known as *source inputs*) to exercise diverse program behaviors with no need to prepare oracles for individual inputs [3]. MT has achieved success in detecting faults for various software, such as compilers [4, 5], databases [6, 7], machine translation services [8, 67], and question answering systems [9, 10].

Recently, the work (MR-Scout) reports that developers often encode domain knowledge in test cases that exercise MRs. However, over 70% of 11,000 MR-encoded test cases (MTCs) in the dataset do not contain explicit input relations.

Instead, developers often hard-code the source and follow-up inputs. Figure 4.1a shows an MR-encoded test case intended to have the follow-up input (`dateB`) one day after the source input (`dateA`), but it simply hard-codes the two inputs. Without an explicit input transformation program, follow-up inputs cannot be directly generated from automatically generated source inputs. This limitation hinders the reuse of valuable encoded MRs to achieve automated MT and enhance test adequacy. **This work aims to overcome this obstacle by inferring an explicit input relation from a given test case with its hard-coded input pairs.** Specifically, our goal is to construct an input transformation function that turns a source input into a follow-up input as shown in Figure 4.1b. With such input transformations, these encoded MRs can apply to a wider range of test inputs to test SUTs more exhaustively (Figure 4.1c).

This task can be viewed as a programming by example (PBE) problem, where the aim is to synthesize a transformation function that turns a given source input into the corresponding follow-up input. The challenge lies in *correctly interpreting the contextual information, such as the relationship between hard-coded input pairs, output relations, and the properties of the SUT.* Moreover, with only one pair of source and follow-up inputs available as an example [3], there is a risk of generating program overfitted to the given example instead of realizing the true intention, as noted in existing PBE studies [68–70]. Therefore, *effectively leveraging contextual* *information is crucial to guide PBE and generate a generalizable input transformation* that aligns with the *semantic* of the encoded MR, ensuring it applies to all potential source inputs with the corresponding output relation.

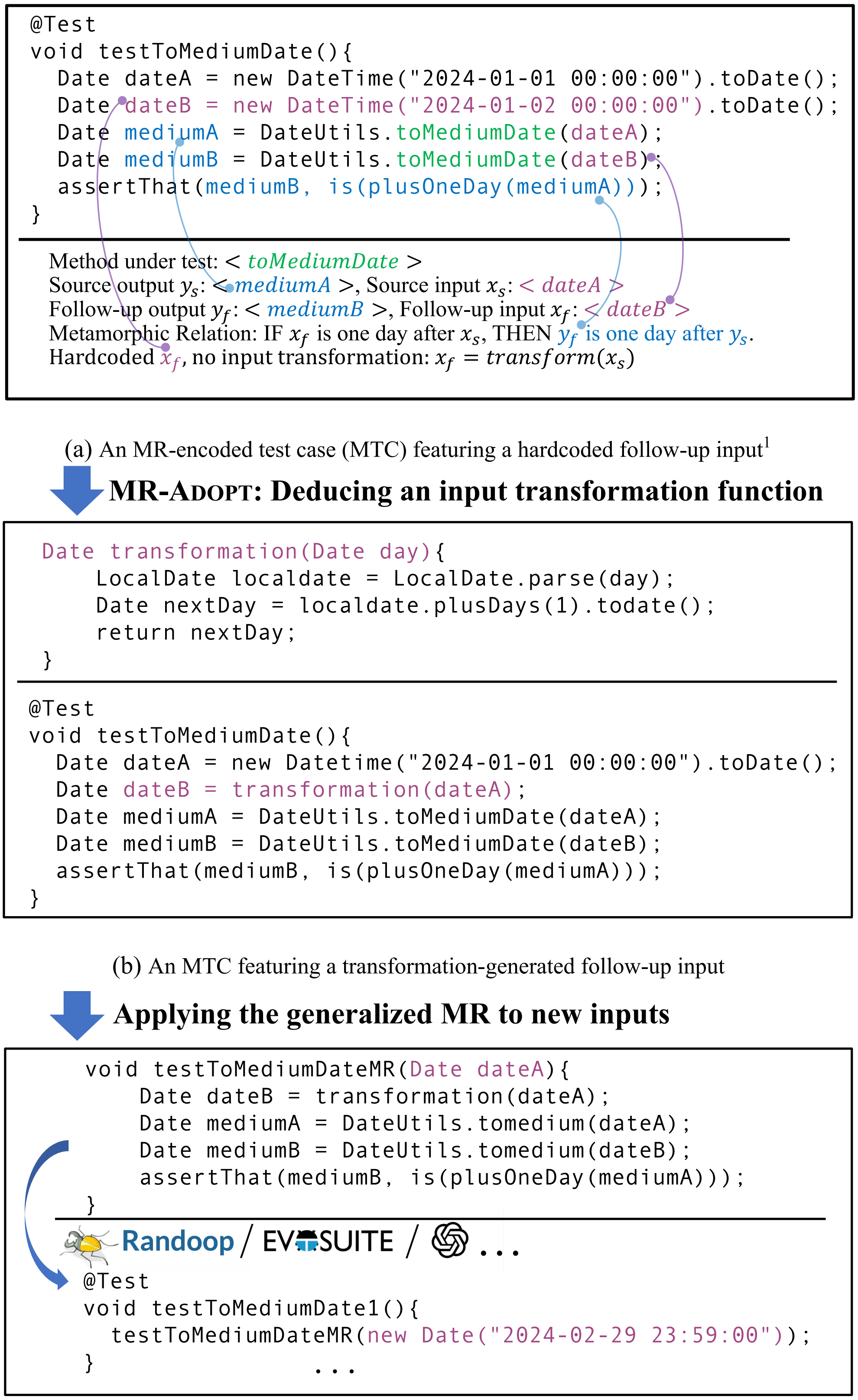
This work proposes MR-Adopt, an approach that leverages large language models (LLMs) to automatically generate input transformation functions for MRs encoded in existing test cases. Trained on extensive code corpora from various domains, LLMs have demonstrated effectiveness in code understanding [71–73] and generation [74–76]. Thus, LLMs have the potential to understand contextual information and generate code based on such information. The insight is to leverage the code understanding ability of LLMs to mine the intention of MR and input relation from the hard-coded test inputs and SUT's function, and take advantage of their code generation ability to produce good input transformation code. This work proposes three designs to harness LLMs' abilities.

Firstly, this work observes that directly providing LLMs with contextual information only results in around 50% generalizable transformations (Section 4.3.5). This is unsatisfactory. To address this, this work needs a design that allows LLMs to effectively express the input relation inferred from the hard-coded inputs and generate transformation code. *To realize this goal,* this work designs MR-Adopt with two phases. In *Phase1*, LLMs perform analogical reasoning [77, 78] on the hardcoded source-followup input pairs to infer new input pairs that obey the same input relation. In *Phase2*, LLMs generate an input transformation function based on (i) the input pair hard-coded by developers and (ii) additional input pairs generated by LLMs in *Phase1*. This design not only enables LLMs to generate code in their familiar setup (where a task description and several examples are provided) [79], but also mitigates the above-mentioned overfitting issue due to the limited number of examples.

Secondly, this work found that LLMs often generate task-irrelevant code segments, of which some are even faulty. For example, when tasked with generating a test input, an LLM might include an incorrect assertion statement. MR-Adopt addresses this by refining the LLM-generated code through data-flow analysis, extracting only the relevant code for the given task (i.e., additional input pairs and input transformation generation).

Thirdly, to mitigate the errors in the relevant codes generated by LLMs, MR-Adopt leverages the developer-written output relations (i.e., assertions) in MTCs as oracles to verify the generated test pairs. MR-Adopt further employs additional inputs to select the most generalizable input transformation as the result.

[1] This MR-encoded test case is crafted from `org.hisp.dhis.util` in project dhis2-core, where long format date is "`yyyy-mm-dd hh:mm:ss`" and medium format date is "`yyyy-mm-dd`".



```
@Test
void testToMediumDate(){
  Date dateA = new DateTime("2024-01-01 00:00:00").toDate();
  Date dateB = new DateTime("2024-01-02 00:00:00").toDate();
  Date mediumA = DateUtils.toMediumDate(dateA);
  Date mediumB = DateUtils.toMediumDate(dateB);
  assertThat(mediumB, is(plusOneDay(mediumA)));
}
```

Method under test: < *toMediumDate* >
Source output y_s: < *mediumA* >, Source input x_s: < *dateA* >
Follow-up output y_f: < *mediumB* >, Follow-up input x_f: < *dateB* >
Metamorphic Relation: IF x_f is one day after x_s, THEN y_f is one day after y_s.
Hardcoded x_f, no input transformation: $x_f = transform(x_s)$

(a) An MR-encoded test case (MTC) featuring a hardcoded follow-up input[1]

```
 Date transformation(Date day){
     LocalDate localdate = LocalDate.parse(day);
     Date nextDay = localdate.plusDays(1).todate();
     return nextDay;
 }
```

```
@Test
void testToMediumDate(){
  Date dateA = new Datetime("2024-01-01 00:00:00").toDate();
  Date dateB = transformation(dateA);
  Date mediumA = DateUtils.toMediumDate(dateA);
  Date mediumB = DateUtils.toMediumDate(dateB);
  assertThat(mediumB, is(plusOneDay(mediumA)));
}
```

(b) An MTC featuring a transformation-generated follow-up input

```
void testToMediumDateMR(Date dateA){
    Date dateB = transformation(dateA);
    Date mediumA = DateUtils.tomedium(dateA);
    Date mediumB = DateUtils.tomedium(dateB);
    assertThat(mediumB, is(plusOneDay(mediumA)));
}
```

```
@Test
void testToMediumDate1(){
  testToMediumDateMR(new Date("2024-02-29 23:59:00"));
}                         ...
```

(c) Metamorphic testing by integrating an MR with diverse source inputs

Figure 4.1: Overview of MR-Adopt for Metamorphic Testing

This work evaluated MR-Adopt with 100 developer-written test cases that encode MRs. The results show that MR-Adopt can generate compilable input transformations for 95 MRs, where 72 can generalize to all potential source inputs prepared in our evaluation. MR-Adopt generates 17.28% more compilable transformations and 33.33% more generalizable transformations than directly prompting GPT-3.5. Besides, MR-Adopt-generated transformations produce follow-up inputs for 91.21% source inputs, representing a 122.10% improvement over GPT-3.5 in generating follow-up inputs. Our ablation study indicates that all three designs (i.e., additional input pairs, date-flow analysis based refinement, and output-relation based validation) contribute to MR-Adopt's overall performance, with validation and additional input pairs having the most impact. Furthermore, incorporating MRs with input transformations and new source inputs leads to 10.62% and 18.91% increases in line coverage and mutation score on top of developer-written test cases, demonstrating the practical usefulness of MR-Adopt-generated transformations in enhancing test adequacy.

This work makes the following contribution:

- To the best of the author's knowledge, this work is the first to generate input transformations for MRs encoded in test cases. With the generated input transformations, more encoded MRs can be reused to enhance the test adequacy of SUTs.
- This work proposes MR-Adopt, an LLM-based approach to deduce input transformation functions. By generating multiple example input pairs, MR-Adopt mitigates overfitting and produces generalizable transformations. It also incorporates a code refinement strategy based on data-flow analysis and a validation strategy to mitigate the faulty irrelevant code generated by LLMs. This design can be applied to other code generation tasks.
- This work extensively evaluates MR-Adopt's effectiveness in generating input transformations. Results show that MR-Adopt can generate effective input transformations, where 72% input transformations are generalizable to all prepared source inputs. When integrated with these transformations, the encoded MRs increase line coverage by 10.62% and mutation score by 18.91%.
- This work builds a dataset of 100 encoded MRs dated after 01-April, 2023, and released it with our replication package on the website [80].

4.1 Problem Formulation

MR-encoded test cases (MTCs), introduced by Xu et al. [3], are test cases encoded domain-specific knowledge that suggests useful MRs. These MTCs are prevalent, with over 11,000

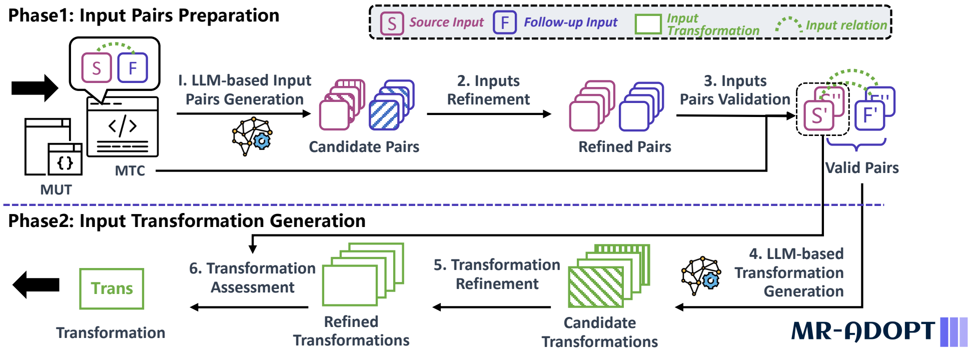

Phase1: Input Pairs Preparation
S Source Input
F Follow-up Input
Input Transformation
Input relation
S
F
MUT
MTC
I. LLM-based Input Pairs Generation
Candidate Pairs
2. Inputs Refinement
Refined Pairs
3. Inputs Pairs Validation
S''
F''
S'
F'
Valid Pairs
Phase2: Input Transformation Generation
4. LLM-based Transformation Generation
Candidate Transformations
5. Transformation Refinement
Refined Transformations
6. Transformation Assessment
Trans
Transformation
MR-ADOPT

Figure 4.2: An overview of MR-Adopt

identified across 701 open-source projects in their study. An MTC can be considered as an instance of an MR, already implemented with specific source and follow-up input values, invocations of methods under test, and output relation assertions. These output relations are well-coded and serve as oracles, requiring no further refinement to apply to new test inputs. Such encoded MRs can be generalized to new inputs and facilitate automated MT by incorporating automatic input generation techniques.

Consider the example in Figure 4.1, the encoded MR in this test case is: “IF a date x_1 in long format (“yyyy-mm-dd hh:mm:ss”) is one day ahead of another long-format date x_2 ($\mathcal{R}_i$), THEN x_1 in medium format (“yyyy-mm-dd”) should also be one day ahead of medium-format x_2 ($\mathcal{R}_o$)”. The SUT method `toMediumDate` is executed on the source input `dateA` and the follow-up input `dateB` separately, and the corresponding outputs are verified by `assertThat(mediumB, is(plusOneDay(mediumA))`, which implements $\mathcal{R}_o$.

Such an implemented MR instance can be reused and generalized to many new inputs. However, the follow-up input `dateB` is hardcoded as `"2024-01-02 00:00:00"` instead of being generated from `dateA` by an input transformation program. While the $\mathcal{R}_o$ is explicitly coded, the $\mathcal{R}_i$ remains **implicit**, hidden within the specific source and follow-up input values `dateA` and `dateB`. According to Xu et al.’s study, over 70% of MR-encoded test cases lack explicitly coded $\mathcal{R}_i$ (i.e., input transformations). This limitation prevents these MRs from being directly applied to new inputs automatically generated by existing tools, e.g., Evosuite [32] and Randoop [31]. While these tools are proficient in generating diverse source inputs, they cannot generate input pairs that satisfy an input relation.

In this work, we address this limitation by deriving an explicit input relation from a given test case and its hardcoded input pairs. Specifically, our goal is to construct an **input transformation function** that converts a source input into a follow-up input, as shown in Figure 4.1b. With such input transformations, embedded MRs can be reused with a broader range of test inputs (Figure 4.1c) to exercise more SUT’s behaviors, thereby enhancing test adequacy. Additionally, these developer-written MRs serve as reliable oracles for new test generation.

4.2 Approach

Figure 4.2 presents an overview of MR-Adopt. It takes a pair of source and follow-up inputs, along with its context (i.e., an MR-encoded test case (MTC) and methods under test (MUT)), and outputs an input transformation function. MR-Adopt works in a two-phase pipeline. In the first phase, it generates additional source-follow-up input pairs and uses them as examples to better describe the input relation, which provides useful guidance for the generation of input transformations. In the second phase, it generates input transformation functions based on these example pairs. This setup, familiar to LLMs for code generation tasks, includes a task description and several examples [79], providing more information to effectively guide LLMs in generating generalized transformations.

In each phase, MR-Adopt employs generation, refinement, and validation procedures. In *Phase1*, MR-Adopt first leverages LLMs to generate candidate test input pairs, then refines them based on data-flow analysis to exclude irrelevant code that can contain errors, and finally filters valid input pairs based on output relation assertions. In *Phase2*, MR-Adopt leverages LLMs to generate candidate input transformations based on the input pairs from *Phase1*. These candidate transformations are then refined by removing irrelevant code elements and adding dependencies, and assessed by applying them to additional source inputs. Ultimately, MR-Adopt outputs the most generalizable transformation function.

4.2.1 *Phase 1:* Input Pair Preparation

Input Pair Generation. MR-Adopt uses an LLM to produce new source-followup input pairs by imitating a given input pair within the context of an existing MTC (which includes the input pair and developer-written assertions checking the output relation) and corresponding methods under test.

Following the idea of the Chain of Thought strategy [81], MR-Adopt prompts an LLM in two steps: first to generate source inputs, and then to generate the corresponding follow-up inputs. This step-by-step approach is adopted because the preliminary experiments found that LLMs perform better when generating source and follow-up inputs sequentially rather than generating entire input pairs at once. Our source input generation prompt follows recent practices [82, 83], and includes (i) a system message about the role of a Java expert and the task to generate test inputs, (ii) the code of methods under tests (MUTs), (iii) the code of the MR-encoded test case (MTC), and (iv) the output format. Such a prompt provides necessary contextual information (ii

Listing 3 Examples of LLM-generated source inputs

~~~
## New source input 1:
```java
    Date dateA = new DateTime("2023-12-31 23:59:59").toDate();
```
## New source input 2:
```java
    DateTime dateTimeA = new DateTime("2024-11-30 23:59:59");
    Date dateA = dateTimeA.toDate();
```
## New source input 3:
```java
    Date dateA = new DateTime("2024-01-01 00:00:00").toDate();
```
        ...(other inputs are ommitted)...
~~~

and iii) and task description (i and iv) for generating source inputs. Detailed prompt templates and examples are available on MR-Adopt's website [80]. Listing 3 shows several example source inputs generated by GPT-3.5 with this prompt.

The follow-up input generation prompt is similar to the source input prompt, with the key difference being the addition of previously generated example source inputs to guide the creation of follow-up inputs. MR-Adopt also adjusts the task description and output format to instruct LLMs to generate source-followup input pairs, using the original pair in the MTC as a sample. There is a trade-off between providing enough examples to guide the following generation of generalizable transformations and maintaining efficiency in terms of time and cost. To balance these factors, MR-Adopt uses five examples. Following recent studies' nucleus sampling [76, 84], MR-Adopt repeats the above generation process five times with a temperature setting of 0.2 [85, 86]. Listing 4 shows several example input pairs.

Input Pair Refinement. Consistent with observations in existing studies [87], the experiments found that LLM-generated codes often include irrelevant segments, which may introduce errors. For instance, the second input pair in Listing 4 includes an assertion statement `assertEquals(dateTimeB, is(plusOneDay(dateTimeA)));` alongside the desired input pair code. This assertion is irrelevant to constructing `dateA` or `dateB` and incorrectly calls the method `plusOneDay(Date date)` with a `DateTime` object, resulting in a type mismatch exception.

Our task focuses on constructing source and follow-up inputs. To exclude such irrelevant code and bypass unnecessary errors, MR-Adopt performs a data-flow analysis on the LLM-generated code and build a dependency graph. MR-Adopt constructs data dependency graphs for source/follow-up inputs by first identifying statements that define source/follow-up inputs, extracting variables used for these definitions, and then recursively tracing these variables' dependencies. Finally, MR-Adopt isolates the dependent statements related to the source and follow-up inputs and removes other statements. For example, in the second input pair of

Listing 4 Examples of LLM-generated input pairs

```
1     The inferred relation between the source input `dateA` and the follow-
2     up input `dateB` is that `dateA` is one day ahead of `dateB`.
3     Let's construct the follow-up inputs for the new source inputs:
4     ## New input pair 1:
5     ```java
6         Date dateA = new DateTime("2024-12-31 23:59:59").toDate();
7         Date dateB = new DateTime("2025-01-01 23:59:59").toDate();
8     ```
9     ## New input pair 2:
10    ```java
11        DateTime dateTimeA = new DateTime("2024-11-30 23:59:59");
12        Date dateA = dateTimeA.toDate();
13        DateTime dateTimeB = new DateTime("2024-12-01 23:59:59");
14        Date dateB = dateTimeB.toDate();
15        assertEquals( dateTimeB, is(plusOneDay(dateTimeA)) );
16    ```
17    ## New input pair 3:
18    ```java
19        Date dateA = new DateTime("2024-01-01 00:00:00").toDate();
20        Date dateB = new DateTime("2025-01-01 00:00:00").toDate();
21    ```
22                ...(other input pairs are ommitted)...
```

Listing 4, the source input dateA and follow-up input dateB depend on objects dateTimeA and dateTimeB, respectively. Thus, the statements (Lines 11-14) for constructing dateA, dateTimeA, dateB, and dateTimeB are considered relevant, while the assertion statement (Line 15) is excluded. Finally, MR-Adopt retains only the statements relevant to constructing source and follow-up inputs, excluding all other irrelevant statements from the LLM-generated code.

Input Pair Validation. The previous refinement step removes the irrelevant code segments generated by LLMs and results in candidate source-followup input pairs. However, a pair of inputs still can be *invalid* if they violate the input relation of an encoded MR. For example, the third input pair shown in Listing 4 is an invalid test pair. The input relation of the embedded MR is that "dateA is one day ahead of dateB", while an LLM generates an input pair of "2024-01-01" and "2025-01-01". Such test case pairs do not align with the intended input transformation and can mislead the generation of transformations. To address this, this work proposes a method to discard such invalid input pairs.

MR-Adopt uses the output relation of an encoded MR to validate LLM-generated input pairs. Specifically, MR-Adopt executes SUT on generated input pairs and checks the outputs against the output relation, which is an explicit reusable code in the MTC, i.e., the developer-written assertions (Line 7 in Figure 4.1a). For each input pair, if its outputs of invoking methods under test on the inputs pass the developer-written assertions, MR-Adopt considers it a valid input pair. As shown in Listing 5, if the outputs mediumA and mediumB pass the assertion (Line 9), the inputs dateA and dateB are considered valid. This step aims to filter out invalid input pairs

Listing 5 Validating an LLM-generated input pair

```java
@Test
void testToMediumDate(){
    // LLM-generated new source input
    Date dateA = new DateTime("2024-12-31 23:59:59").toDate();
    // LLM-generated corresponding follow-up input
    Date dateB = new DateTime("2025-01-01 23:59:59").toDate();
    Date mediumA = DateUtils.toMediumDate(dateA);
    Date mediumB = DateUtils.toMediumDate(dateB);
    assertThat(mediumB, is(plusOneDay(mediumA)));
}
```

Listing 6 An example of output format in the prompt

~~~
# OUTPUT FORMAT
Generate the transformation function by complementing the following
code skeleton.

    ```java
    public static Date transformation(Date day)  {
        // TODO
        Date nextDay =
        return nextDay;
    }
    ```
~~~

generated by LLMs from the example set. It could discard some source-followup input pairs that match the input relation in fact. Factors such as the bugs in a non-regression SUT may lead to false violations and mistaken deletions of these pairs. However, the goal of the first phase is to prepare examples that give more information about the input relation for the second phase. Thus, it does not require *complete* source-followup input pairs.

4.2.2 *Phase 2:* Transformation Generation

Transformation Generation. In this step, MR-Adopt instructs an LLM to generate candidate input transformation functions for an encoded MR by providing example source-followup input pairs. The examples include the original hard-coded pair and additional pairs generated in *Phase1*. The prompt for transformation generation is similar to the input pair generation prompt (Section 4.2.1), consisting of (i) a system message, (ii) the code of MUT, (iii) example input pairs, (iv) the code of an MTC, and (v) the output format. The difference is that the task shifts from generating source-followup input pairs to generating input transformation functions, whose parameter list and return type are already specified. Detailed prompt template and samples are available on MR-Adopt's website [80].

Listing 6 shows the output format specified in the prompt, which defines the skeleton of the input transformation function to generate. It includes the function name, parameter (i.e., source input) types and names, and type of the return value (i.e., follow-up input) [2]. Following recent

[2] For MRs with multiple follow-up inputs, the return type is a list of objects.

studies' nucleus sampling [76, 84], for each MR, MR-Adopt instructs an LLM to generate one input transformation function, and repeats the generation process five times with a temperature setting of 0.2 [85, 86]. Finally, five candidate transformation functions can be generated.

MR-Adopt extracts the generated functions by identifying code blocks wrapped with ``` and extracting the code that matches the given transformation function skeleton. This ensures the generated code conforms to the required format and can be easily integrated into given MR-encoded test cases.

Transformation Refinement. Similar to the situation discussed in Section 4.2.1, LLM-generated transformation functions can contain irrelevant code, some of which can cause errors (e.g., invoking non-existing APIs). To address this issue, MR-Adopt constructs data dependency graphs for follow-up inputs by first identifying statements that define these inputs. It then extracts the variables used in these definitions and recursively traces their dependencies. Statements involved in the dependency graph are considered relevant and retained, while others are considered irrelevant and excluded. As the example shown in Listing 7, the follow-up input `nextDay` depends on `localDate`, which further depends on `day`. Statements constructing `nextDay` and `localDate` are retained, while irrelevant statements such as `Date dayAfter=day.after(1)` are excluded.

Listing 7 An example of LLM-generated transformation

~~~
The transformation function can be implemented as follows:

```java
public static Date transformation(Date day)  {
    Date dayAfter = day.after(1); // non-exisitng API and irrelevant
    LocalDate localdate = LocalDate.parse(day);
    Date nextDay = localdate.plusDays(1).todate();
    return nextDay;
}
```
~~~

After excluding irrelevant code, MR-Adopt analyzes and imports dependencies needed by the generated transformation function. Using JavaParser [88], MR-Adopt identifies dependent class names through syntax analysis. It then retrieves potential classes defined or imported in source and imports those whose names match the dependent classes. For example, the internal class `LocalDate` Listing 7 will be imported. Although matching dependent classes by name can be inaccurate if LLM-generated code uses incorrect names or third-party libraries not imported in the project, our experimental observations indicate that this issue is minor. Most dependent classes are derived from the context provided to LLMs, such as the MUT or MTC code.

Transformation Assessment. After refining candidate transformations, MR-Adopt further assesses their quality by applying them to new source inputs. In this step, MR-Adopt leverages

Listing 8 Validating an LLM-generated input transformation with a new source input

```
1  @Test
2  void testToMediumDate() {
3      // new source input
4      Date dateA = new Date("2024-02-01 00:00:00");
5      // invoking generated transformation on new source input
6      Date dateB = transformation(dateA);
7      Date mediumA = DateUtils.toMediumDate(dateA);
8      Date mediumB = DateUtils.toMediumDate(dateB);
9      assertThat(mediumB, is(plusOneDay(mediumA)));
10 }
```

Listing 9 An example of LLM-generated transformation

```
public static Date transformation(Date day) {
    int dayValue = day.getDate();
    int monthValue = 1; // set the month to 1 since we don't know which month the input
     ↪ represents
    int yearValue = day.getYear();
    Date nextDay = new Date(year, month, day +1);
    return nextDay;
}
```

new source inputs generated in Section 4.2.1 to assess the generalizability of candidates and then selects the most generalizable one.

Specifically, MR-Adopt uses new source inputs as test inputs and employs developer-written assertions (i.e., output relation assertions) as test oracles. A transformation is considered applicable to a given source input if (a) the input transformation function can successfully generate a corresponding follow-up input without throwing exceptions, and (b) the outputs from executing the methods under test pass the developer-written assertions. For example, in Listing 8, given the source input `dateA`, if the follow-up input `dateB` is successfully generated and the outputs `mediumA` and `mediumB` pass the assertion (Line 9), MR-Adopt considers the transformation applicable to input `dateA`. Conversely, Listing 9 shows a failing transformation that only works for January dates. MR-Adopt assesses all candidate transformation functions using both new and the original source input. It then selects the most generalizable transformation that applies to the most inputs. In case of a tie, MR-Adopt returns the first generated one as the result.

4.3 Evaluation

4.3.1 Research Questions

Our evaluation aims to answer the following research questions:

- **RQ5:** *How effective is* MR-Adopt *in generating input transformations?* This RQ compares the quality of the input transformation functions generated by MR-Adopt and baselines to evaluate the effectiveness of MR-Adopt in generating generalizable input transformations for

MRs encoded in test cases.

- **RQ6:** *How effective are* MR-Adopt-*generated input transformations in constructing follow-up inputs, compared with LLMs?* This RQ investigates the benefits of generating input transformation functions, by comparing the quality of follow-up inputs produced by these functions versus those directly generated by LLMs.
- **RQ7:** *What is the contribution of each component in* MR-Adopt*?* This RQ performs an ablation study to reveal how each component contributes to generating input transformations.
- **RQ8:** *How useful are encoded MRs in enhancing test adequacy with the generated input transformations?* With input transformations generated by MR-Adopt, more encoded MRs can be reused with new inputs to test more behaviors of SUT. This RQ investigates the usefulness of such encoded MRs in improving test adequacy, demonstrating the usefulness of generating input transformation.

4.3.2 Dataset

MR-encoded test cases (MTCs). This work followed Xu *et al.* [3] to collect high-quality Java projects with at least 200 stars over GitHub. Besides, this work further excluded the projects created before 01-April 2023 to prevent the experimental LLMs from having potentially learned the code during training, thereby reducing the potential for data leakage [85]. Finally, this work collected 2,007 MTCs from qualified projects. From these MTCs, this work retained test cases that (i) can be successfully compiled, (ii) can be successfully executed (i.e., passing developer-written assertions), and (iii) contain MRs associated with exactly two method invocations (one for the source input and one for the follow-up input). The third criterion excludes complex and less common MRs involving multiple input groups [3]. MRs with exactly two invocations constitute the majority (65%) of MTCs [3]. Handling MRs with more than two invocations presents significant challenges in identifying source and follow-up inputs. This problem is nontrivial and remains an important future work. Finally, this work obtained 180 MTCs, including 54 with explicit input transformations written by developers and 126 without such transformations, consistent with the distribution reported by Xu *et al.* [3].

Generation Tasks and Ground Truths. Based on the collected 180 MTCs, this work prepared a dataset containing (i) 100 MTCs without input transformations as tasks, and (ii) corresponding input transformation functions as ground truths. The preparation process is as follows. Firstly, this work tried to utilize all 54 MTCs with ground truths, i.e., developer-written input transformations. For each MTC, this work executed the input transformation on the hardcoded source input to obtain the follow-up input. This work prepared a task by replacing the developer-written transformation with the hardcoded follow-up input. Some MTCs whose follow-up input cannot be hardcoded are excluded. For example, an MR for a text render class is "the width of a text (source input) should not be greater than its bold version (follow-up input)". The follow-up input (bold text) can only be generated by a method `bold()`, which is a developer-used transformation program. Finally, this work built 36 tasks from 36 MTCs with developer-written transformations.

Next, the author and collaborators manually constructed input transformation functions for MTCs lacking developer-written transformations. Specifically, 64 out of 126 MTCs without input transformations were randomly selected as tasks. For each task, one PhD student in the same research group examined the SUT and its underlying MRs and then created a transformation function applicable to the original source input and generalizable to new valid source inputs. Another PhD student reviewed these transformations, and any disagreements were discussed and resolved with consensus. This process took approximately 200 human hours. Details of this dataset can be found on MR-Adopt's website [80].

4.3.3 Environment and Large Language Models

Our experiments were conducted on machines with three RTX4090 GPUs, dual Intel Xeon E5-2683 v4 CPUs, and 256 GB RAM.

The large language models used in our evaluation include GPT-3.5 from OpenAI [89] and three open-source code models: Llama3-8B [90] from Meta, Deepseek-coder-7b [91] from DeepSeek, and CodeQwen1.5-7B-Chat [92] from Alibaba. This work uses these LLMs since they are popular state-of-the-art [93] code models in well-known LLM families and deployable at our machines.

4.3.4 Source Input Preparation

To evaluate the generated input transformations, this work needs new valid source inputs as a "test set". Automatic test input generation techniques (such as Evosuite [32] or Randoop [31]) can be employed to prepare source inputs. However, this work found that these tools often fail to generate test inputs for many MRs. This is because over 50% of experimental MRs' inputs are user-defined complex objects with complicated preconditions and environments, which are challenging for tools like Evosuite to handle. This aligns with Xu et al.'s observation [3].

Recent studies show that LLMs are good test input generators [82, 83]. In this study, this

Table 4.1: Effectiveness of MR-Adopt in generating input transformations for 100 MRs encoded in test cases

Metric (# Trans.)	**Direct Prompting**			**MR-Adopt**		
	Llama3	*Deepseek*	*GPT-3.5*	*Llama3*	*Deepseek*	*GPT-3.5*
compilable	79	80	81	86 (+8.86%)	89 (+11.25%)	**95 (+17.28%)**
>0% generalizable	69	72	69	77 (+11.59%)	82 (+13.89%)	**83 (+20.29%)**
>75% generalizable	64	67	63	74 (+15.66%)	80 (+19.40%)	**81 (+28.57%)**
100% generalizable	57	60	54	68 (+19.30%)	71 (+18.33%)	**72 (+33.33%)**

# *n% generalizable:* the number of generated input transformations applicable to at least *n*% of source inputs.

work employed an LLM (Qwen) to generate new source inputs for evaluating transformations, while other experimental LLMs were used to generate transformations. As a reminder, to mitigate circular evaluation, this work employed different LLMs for preparing the "test set" and for generating input pairs and transformation functions in MR-Adopt. This work reused the prompt template from MR-Adopt's *Phase1*. Qwen was instructed to generate five source inputs at a time, and the process was repeated ten times with a 0.2 temperature setting to produce more source inputs.

For the 100 experimental MRs, Qwen generated a total of 5,355 new source inputs. This work first filtered out 3,058 duplicate inputs using string matching. Next, this work identified valid source inputs by executing them on the corresponding ground truth transformations. A source input is considered valid only if the ground truth transformation successfully generates a follow-up input, and the outputs of this source input and corresponding follow-up input pass the developer-written assertions (R_o). Qwen failed to generate a new valid source input for 5 MRs whose inputs are complex objects and have strict domain-specific constraints. Finally, this work collected 1,366 valid source inputs, averaging 14.37 per MR.

4.3.5 RQ5: Effectiveness of MR-Adopt

Experiment Setup. This RQ inspects MR-Adopt's effectiveness in generating input transformation functions by examining their comparability and generalizability to new source inputs.

Baselines. To the best of the author's knowledge, no existing approach generates input transformation functions for MRs across different domains. Given the proven effectiveness of LLMs in code and test generation, this work set *directly prompting LLMs* as a baseline. Specifically, this work directly prompted GPT-3.5-turbo-0125, Llama3-8B-Instruct, and Deepseek-coder-7b-instruct-v1.5 (shorten as GPT-3.5, Llama3, and Deepseek, respectively). The template is similar to MR-Adopt's and available at [80]. The knowledge cut-off dates for these models are September 2021 [94], March 2023 [95], and March 2023 [96], respectively, before the creation date of our dataset's MTCs (Section 4.3.2),

Configuration of Baseline LLMs. Following recent studies [76], this work used the nucleus sampling [84] and repeated the generation process five times for each task with a temperature setting of 0.2 [85, 86], and selected the best result for comparison. The configuration of MR-Adopt was introduced in Section 4.2.2.

Metrics. For this RQ, this work introduced two metrics: (i) # *compilable transformations:* the number of generated input transformations that can successfully compile, and (ii) # *n% generalizable transformations:* the number of generated input transformations applicable to at least *n*% of source inputs prepared in Section 4.3.4 ($n = 0, 75, 100$ representing at least one, upper-quartile, and all inputs, respectively). A transformation t is considered *applicable* to a source input x_s if t generates a follow-up input x_f for x_s, so that a *correct* SUT does not violate the output relation on the input pair $<x_s, x_f>$.

Result. As shown in Table 4.1, MR-Adopt effectively produced many compilable input transformation functions that well generalize to prepared source inputs. We found that MR-Adopt works best with GPT-3.5. Specifically, using GPT-3.5 (the last column), MR-Adopt produced compilable transformations for 95 out of 100 MRs, with 72 of these transformations effectively applied to *all* prepared source inputs. MR-Adopt also works well with Llama3 and Deepseek, generating 68 and 71 (100%) generalizable transformations, respectively. Besides, some generated transformations generalize well to some, but not all, source inputs prepared in our experiment. Specifically, with GPT-3.5, 83 out of 95 compilable transformations applied to at least one source input, and 81 of them applied to more than 75% of the prepared source inputs. Similar results were found with Llama3 and Deepseek. This work considered these transformations generated by MR-Adopt *still useful* to some extent, as they successfully prepare some valid input pairs. Upon further analysis, this work found that their limitations could potentially be addressed with more comprehensive prompts to handle corner cases. LLM-generated transformations effectively handle common cases but struggle with edge cases. For example, an ideal transformation would generate a higher version string in any scenario (e.g., transforming `"1.0-A1"` to `"1.0-B1"`), but the LLM-generated transformation relies on a 'Major.Minor.Revision' convention (e.g., `"1.0.1"`) and fails with cases like `"1.0-A1"`.

There were 5, 14, and 11 transformations generated by MR-Adopt with GPT-3.5, Llama3, and Deepseek, respectively, that failed to compile. The main reasons include: (i) the generated transformations invoke non-existing methods to generate the follow-up input and (ii) they invoke inaccessible APIs due to permission restrictions (e.g., private methods). Additionally, the compilable but not generalizable transformations were primarily due to *Phase1* failing to generate valid input pairs for these MRs, leading to LLM-generated transformations overfitted to the

Table 4.2: Effectiveness of MR-Adopt's transformations in constructing follow-up inputs for 1366 source inputs

MR-Adopt	Llama3	Deepseek	GPT-3.5	**Improvement**
1246	697	724	597	+72.10%~**+108.71%**
MR-Adopt	Llama3$^+$	Deepseek$^+$	GPT-3.5$^+$	**Improvement**
1246	770	737	708	+61.82%~**+75.99%**

$^+$ means incorporating MR-Adopt's input refinement procedure for LLMs' answers.

given input pair.

This work also compared MR-Adopt's performance (columns 5-7) with the baseline of directly prompting LLMs (columns 2-4). Although the output relation encoded in MTC was provided in prompts for baselines to aid transformation generation, MR-Adopt still generated more compilable transformations. This improvement is due to MR-Adopt's code refinement and assessment strategies. Moreover, MR-Adopt demonstrates substantial improvements in generating transformations that are >75% and 100% generalizable, with increases of 15.66% to 28.57% and 18.33% to 33.33%, respectively. This suggests the effectiveness of preparing more examples for LLMs and the benefits of MR-Adopt's refinement and selection strategies.

Answer to RQ5: MR-Adopt significantly outperforms the baseline LLMs across all metrics. Compared to directly prompting LLMs, MR-Adopt achieves 18.33%~33.33% improvement in generating 100% generalizable input transformations.

4.3.6 RQ6: Effectiveness of Input Transformations

Experiment Setup. This RQ examined the quality of follow-up inputs produced by input transformations generated by MR-Adopt. This work set LLMs as the baselines because they are off-the-shelf black-box transformations that can generate follow-up inputs given source inputs, as introduced in Section 4.2.1. This work also included LLMs enhanced with MR-Adopt's refinement procedure (marked with $^+$) for comparison. This can reflect the effectiveness of MR-Adopt's refinement for input pairs preparation (Section 4.2.1).

Metric. This work generated follow-up inputs by feeding the 1,366 prepared source inputs (Section 4.3.4) to input transformations generated by MR-Adopt and the vanilla LLM baselines. To compare the qualities of the follow-up inputs produced by the MR-Adopt-generated transformations and the baselines, this work used the number of *valid* follow-up inputs as the metric. Similar to Section 4.3.5, this work considers a follow-up input x_f *valid* if it and its corresponding source input can pass developer-written output relation assertions.

Result. As shown in Table 4.2, when built with GPT-3.5, input transformation functions generated by MR-Adopt produced valid follow-up inputs for 1246 out of 1366 (91.22%) source inputs. The high validity rate demonstrated that MR-Adopt contributed to abundant useful source-followup input pairs.

In comparison, three vanilla LLMs only generated valid follow-up inputs for 697 (51.02%), 724 (53.00%), and 597 (43.70%) source inputs, respectively. MR-Adopt surpassed them by 72.10%-108.71%. LLMs enhanced with MR-Adopt's input refinement procedure (marked with $^+$) worked better than the vanilla LLMs. This indicates the usefulness of our design to refine the LLM-generated test inputs (Section 4.2.1). Meanwhile, MR-Adopt's transformations still outperformed the enhanced LLMs by generating 61.82% more valid follow-up inputs than Llama3$^+$, 69.06% more than Deepseek$^+$, and 75.99% more than GPT-3.5$^+$. This significant performance gap highlights the effectiveness of MR-Adopt's transformation functions compared to the state-of-the-art LLMs. It also evidenced the usefulness of our idea to codify the input transformation by leveraging the code understanding and generation abilities through the two-phase pipeline and preparation-refinement-validation process.

This work also summarized two major limitations of using vanilla LLMs as black-box transformations based on our observation. Firstly, LLMs can generate a follow-up input with a wrong value, which is similar to the case in Listing 4. Another limitation is that LLMs often fail to capture the constraints between multiple arguments of the follow-up input. For instance, consider a method `deserial(data, size)` to deserialize an `ArrayList` `data` with a given `size`. The `size` should not be greater than the length of `data`. However, LLMs may miss this constraint and generate invalid value for `size`. These issues about value processing could be due to LLMs' limited inference ability. Instead, MR-Adopt asks LLMs to codify the input transformation and uses the code to do calculation and processing, which is recognized as a better way to exert LLMs' abilities [97]. Besides, using LLMs as transformations can be costly since it is needed to request LLMs for each source input. Meanwhile, MR-Adopt uses LLMs to generate transformations for once, and there is no need to query LLMs when using the generated transformations.

Answer to RQ6: MR-Adopt's refinement step can effectively enhance follow-up input generation, with up to 18.59% improvement for GPT-3.5. Additionally, MR-Adopt-generated transformations can effectively generate follow-up inputs for 91.21% source inputs, surpassing GPT-3.5$^+$ by 75.99%.

Table 4.3: Contribution of each component in MR-Adopt

Metrics (# Trans.)	MR-Adopt	v_1: w/o input pairs	v_2: w/o refinement	v_3: w/o assessment
compilable	95	87 (-8.42%)	93 (-2.10%)	95 (0.00%)
>0% generalizable	83	73 (-12.04%)	82 (-1.20%)	70 (-15.66%)
>75% generalizable	81	66 (-18.51%)	75 (-7.40%)	59 **(-27.16%)**
100% generalizable	72	58 **(-19.44%)**	61 **(-15.27%)**	56 (-22.22%)

4.3.7 RQ7: Ablation Study on MR-Adopt

Experiment Setup. This work created three variants v_1, v_2, and v_3 of MR-Adopt by ablating three components to analyze the helpfulness of these designs for generating generalizable input transformations. This work chose MR-Adopt built with GPT-3.5 which achieves the best result in RQ5 (Section 4.3.5). The variants are as follows:

- v_1: **MR-Adopt w/o additional input pairs.** This variant used only one source-followup input pair hard-coded in an MTC to guide the input transformation generation. It did not use additional input pairs generated in MR-Adopt's *Phase1* (Section 4.2.1).
- v_2: **MR-Adopt w/o refinement step.** This variant disabled the refinement step for generated input transformations in MR-Adopt (Section 4.2.2).
- v_3: **MR-Adopt w/o assessment step.** This variant disabled the assessment step for selecting the most generalizable transformations (Section 4.2.2). Instead, it randomly selected one of the compilable transformation functions as the result.

Result. As shown in Table 4.3, removing additional input pairs (v_1) led to a 19.44% decrease in generating 100% generalizable transformations. This suggests that additional input pairs effectively mitigate the overfitting problem caused by the limited examples in PBE [68–70], helping MR-Adopt generate more generalizable transformation.

Similarly, disabling the refinement step (v_2) reduced 15.27% input transformations that generalize to 100% prepared inputs. This indicates that some generated transformations have minor issues and can be refined by excluding irrelevant code. Besides, disabling the assessment step (v_3) decreased 22.22% input transformation generalizable to 100% inputs. This suggests that, even with additional input pairs and refinement, few 100% generalizable transformations can be generated, and random selection may miss them. The assessment step is necessary to rank the most generalizable function.

Answer to RQ7: All three designs contribute to the effectiveness of MR-Adopt in generating generalizable transformations. The assessment procedure contributes the most, and additional example input pairs contribute similarly.

Table 4.4: Enhancement of test adequacy from generalized MR based test cases ($\mathcal{M}$) on top of developer-written ($\mathcal{D}$) and LLM-generated input pairs ($\mathcal{L}$) based test cases

Metrics	**VS.** $\mathcal{D}$			**VS.** $\mathcal{D}$+$\mathcal{L}$		
	$\mathcal{D}$	$\mathcal{D}$+$\mathcal{M}$	**Improve.**	$\mathcal{D}$+$\mathcal{L}$	$\mathcal{D}$+$\mathcal{L}$+$\mathcal{M}$	**Improve.**
Line Coverage	0.2373	0.2625	+10.62%	0.2588	0.2698	+4.25%
Mutation Score	0.1322	0.1572	+18.91%	0.1710	0.1807	+5.67%

4.3.8 RQ8: Usefulness of Input Transformations

Experiment Setup. In this RQ, this work integrated the generated input transformations into MTCs to construct generalized MRs and measured how well such MRs enhanced test adequacy. This demonstrated the practical usefulness of MR-Adopt's transformations in enhancing test adequacy.

New Test Cases Construction. This work applied generalized MRs to the automatically generated source inputs introduced in Section 4.3.4 to obtain a set of new test cases (denoted as $\mathcal{M}$). This work compares such test cases against two baselines: (i) the developer-written test cases (i.e., MTCs) (denoted as $\mathcal{D}$) and (ii) test cases based on the LLM-generated source and follow-up input pairs (denoted as $\mathcal{L}$). Specifically, the prepared source inputs (Section 4.3.4) are combined with valid follow-up inputs generated by Llama3$^+$ which performed the best in RQ6 (Section 4.3.6). Considering generalized MR based test cases and LLM-generated input pairs based test cases are extended from developer-written existing test cases, this work followed Xu *et al.* [3]'s practice to analyze the test adequacy improvement on top of developer-written test cases.

Metrics. This work measured test adequacy using two metrics: (i) Line Coverage – percentage of code lines in target classes executed, and (ii) Mutation Score – percentage of mutants killed by test cases.

Mutation Testing: This work employed Pitest [44] to conduct mutation testing. Each MR only focused on one or two methods under test in the target class. To include the covered lines or killed mutants in the methods intransitively invoked by MR-involved methods for comparison, this work employed Pitest to generate mutants targeting all methods in a target class. Finally, Pitest successfully generated 4,388 mutants for 45 target classes covered by 88 MRs in the dataset (Section 4.3.2). Pitest failed for the other 12 MRs' classes because of environmental issues (e.g., conflict dependencies).

Result. As shown in Table 3.2, compared to developer-written MTCs ($\mathcal{D}$), incorporating new test cases constructed from generalized MR ($\mathcal{D}$+$\mathcal{M}$) increased the line coverage by 10.62% and the mutation score by 18.91%. This suggested that MR-Adopt could enhance the test adequacy by integrating high-quality test oracles (i.e., output relation of the encoded MR) with a diverse set of potential test input pairs of the MR ($\mathcal{M}$). Although the developer-written test inputs hard-coded in MTCs were carefully crafted and invaluable, each typically included one pair of test inputs and could not sufficiently exercise the SUT's behaviors. The new source inputs generated by test generation techniques and the corresponding follow-up inputs enabled by MR-Adopt may reach program states not covered by the hard-coded inputs.

Besides, by analyzing the benefit of using MR-Adopt ($\mathcal{D}$+$\mathcal{L}$+$\mathcal{M}$) over the test suite enhanced by LLM-generated valid input pairs ($\mathcal{D}$+$\mathcal{L}$), we could still observe 4.25% and 5.67% improvements in the line coverage and the mutation score, respectively. This suggested that even if an LLM could act as a black-box transformation to generate some valid source-followup inputs and reach more execution states of SUT, MR-Adopt could generate input transformations that apply to more source inputs and better enhance the test adequacy.

Answer to RQ8: Test cases constructed from generalized MRs could achieve 10.62% and 18.91% increases in the line coverage and mutation score, respectively, demonstrating generalized MRs' practical usefulness in enhancing test adequacy.

4.4 Discussion

4.4.1 Threads to Validity

The author identified potential threats to the validity of our experiments and have taken measures to mitigate them.

Representativeness of Experimental Subjects. A potential threat is whether our evaluation findings can generalize to different projects. To mitigate this threat, this work adopted the criteria from existing studies [3, 49, 50] to select high-quality and well-maintained Java projects as representative subjects (Section 4.3.2) and evaluated our method on these projects. Besides, evaluating LLMs with subjects seen during model training (known as the data leakage issue) will make the findings biased [98]. To mitigate this threat, this work collected MR-encoded test cases created after the training cut-off date of the experimental LLMs, as described in Section 4.3.2.

Representativeness of Experimental LLMs. MR-Adopt depends on LLMs, and this work also uses LLMs as baselines. A potential threat is whether our evaluation findings based on the selected LLMs are representative. To mitigate this threat, this work evaluated our method with LLMs from three well-known LLM families, i.e., GPT-3.5 from OpenAI, Llama3 from Meta, and Deepseek from DeepSeek. They represent the state-of-the-art code LLMs (according to the EvalPlus leaderborad) that can be deployed with the hardware capacity of our machine, as introduced in Section 4.3.3.

Quality of the Experimental Source Inputs. As introduced in Section 4.3.4, this work used an LLM to prepare new source inputs to assess the generalizability of generated input transformations. Low-quality source inputs may threaten the evaluation validity. To mitigate this issue, this work employed another SOTA code LLM (i.e., Qwen) which is not the experimental subject to prepare the source inputs. This work then used the ground truth input transformations to filter out invalid source inputs.

Quality of Ground Truths. Besides directly using developer-written input transformations in MTCs (if available) as ground truths, this work also manually prepared ground truths for MTCs without input transformations. There is a potential threat regarding the quality of our prepared ground truths. To mitigate this threat, two authors (PhD students) proficient at MT and with more than four years of Java programming experience implemented the ground truths after understanding the intention of the SUTs and the encoded MRs. Specifically, a ground truth was developed by one participant and reviewed by the other until a consensus was reached. Furthermore, the developed ground truths are validated against the original source input.

4.4.2 Distinct Advantages of MR-based Tests in Fault Detection

Detecting faults in "non-testable" programs. MR-based tests offer distinct advantages in validating *non-testable programs* whose expected outputs for given inputs are hard to specify [1, 2]. The usefulness of MR-based tests in detecting such faults for such programs has been reported in studies [23, 55, 99, 100]. MR-Adopt targets encoded MRs for testing Java classes. This work provides two examples to illustrate this advantage.

As shown in Listing 10, the class `AES` contains methods to encrypt and decrypt a string. The `encrypt` function includes a fault: it mistakenly encrypts the `secret` argument instead of the intended `source`. However, the expected string literal after encryption is difficult to specify. This makes it difficult to construct an explicit test oracle based on the expected output to effectively validate the behavior of `encrypt`.

This work collected developer-written tests, EvoSuite-generated tests (following Xu et al.'s practice [3]), and LLM-generated tests (following Yuan et al.'s practice and generating tests with GPT-4 [82]). As shown in Listing 11, this work found that the non-MR test (including developer-written and LLM-generated assertions) only checks that the encrypted string is not null or empty.

Listing 10 Faulty class `AES`[3] for encrypting and decrypting

```java
public static abstract class AES {
    public static String encrypt(String source, String secret) {
        ...
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] bytes = cipher.doFinal(toBytes(secret)); // BUG: This should be
          ↪  "toBytes(source)" instead of "toBytes(secret)"
        return Base64.getEncoder().encodeToString(bytes);
    }
    public static String decrypt(String encrypted, String secret) {
    ...
    }
}
```

Listing 11 Non-MR and MR based tests[4] for class `AES`

```java
public void AES_NonMRTest() {
  String source = "!@#!@#!@1fsd"; String secret = "ssdkF$HUy2A#D%kd";
  String encrypted = CipherHelper.AES.encrypt(source, secret);
  // Developer-written non-MR assertion
  assertFalse(Strings.isNullOrEmpty(encrypted));

  // LLM-generated non-MR assertion
  assertNotNull(encrypted); assertFalse(encrypted.isEmpty());

  // EvoSuite-generated non-MR test: no assertion
  String string0 = CipherHelper.AES.encrypt("jlyN5n~l(%", "AES");
  String string1 = CipherHelper.AES.decrypt("n<h!/N*WHF", ")");
}

//Developer-written MR-based test, MR: x=AES.decrypt(AES.encrypt(x))
public void AES_MRTest() {
  String source = "!@#!@#!@1fsd"; String secret = "ssdkF$HUy2A#D%kd";
  String encrypted = CipherHelper.AES.encrypt(source, secret);
  String decrypted = CipherHelper.AES.decrypt(encrypted, secret);
  assertEquals(source, decrypted);
}
```

EvoSuite failed to generate any assertions. However, this test is weak in validating whether the encryption process is correctly implemented. It ensures only the existence of a non-empty output.

In contrast, the developer-written MR-based test validates the encrypted string using an MR: $x = AES.decrypt(AES.encrypt(x))$ — *IF* an input x is encrypted and subsequently decrypted, *THEN* the final result should be x. The MR-based test successfully detects the fault in encrypting the `secret` argument instead of the `source`, while the non-MR tests fail to detect the fault.

Similarly, when testing the `getRegistryCenterTime`[5] function, which is designed to (i) create a new registry entry and return the creation time, or (ii) update an existing entry and return the updating time, it is difficult to determine the expected output because the exact system time

[3] Available in the CipherHelper.java file from the project FlowCI/flow-core-x

[4] Available in the CipherHelperTest.java file from the project FlowCI/flow-core-x

[5] Available in the `ZookeeperRegistryCenter` class within the project apache/shardingsphere-elasticjob

Listing 12 MR-based and non-MR-based tests[6] for `ZookeeperRegistryCenter`

```java
public void GetRegistryCenterTime_nonMRTest() {
  String key = "/_systemTime/current";
  long regCenterTime = zkRegCenter.getRegistryCenterTime(key);
  // Developer-written non-MR assertion
  assertTrue(regCenterTime<=System.currentTimeMillis());
  // LLM-generated non-MR assertion
  assertTrue(regCenterTime > 0L);
}

//Developer-written MR-based test: IF t2=getRegistryCenterTime(key) is called after t1=getRegistryCenterTime(key), THEN t1<t2
public void GetRegistryCenterTime_MRTest() {
  String key = "/_systemTime/current";
  long regCenterTime = zkRegCenter.getRegistryCenterTime(key);
  long updatedRegCenterTime = zkRegCenter.getRegistryCenterTime(key);
  assertTrue(regCenterTime < updatedRegCenterTime);
}
```

depends on when the code is executed.

The non-MR test, including developer-written and LLM-generated assertions, weakly validates if the registration time is greater than 0 and less than the current system time. EvoSuite failed to generate any assertions. Such a weak non-MR test has a significant limitation: it fails to validate whether the function correctly handles the update of existing entries. In other words, it does not explicitly check the chronological order of the successive calls.

In contrast, a developer-written MR-based test validates the registration time based on an MR: *IF* $t_2 = getRegistryCenterTime(key)$ is called after $t_1 = getRegistryCenterTime(key)$, *THEN* $t_1 < t_2$. When these tests are applied to a faulty `getRegistryCenterTime` implementation that returns an unmodified time due to a missing entry update, only the MR-based test can detect this fault, while non-MR tests cannot.

Exercising a wide range of inputs even with a single MR. Another advantage of MT is that *one MR can be applied to a wide range of automatically generated test inputs (known as source inputs) to exercise various program behaviors*.

The experimental results show that useful tests can be constructed from MRs by leveraging MR-Adopt's input transformations and the wide range of generated test inputs. The constructed tests help improve test adequacy with an increase of 10.62% and 18.91% in line coverage and mutation score, respectively. The results are in line with that reported by Xu et al.'s study [3], which observes an increase of 52.10% and 82.80% in line coverage and mutation score, respectively, by MR-based tests over EvoSuite-generated tests. Even compared with the combination of developer-written and EvoSuite-generated tests, MR-based new tests exclusively covered 113

[6] Available in the ZookeeperRegistryCenterQueryWithoutCacheTest.java file from the project apache/shardingsphere-elasticjob

(+6.95%) mutants and killed 88 (+7.93%) mutants. This is because, although EvoSuite can generate many inputs, it fails to generate effective test oracles that can identify incorrect outputs and solve the setups to trigger target programs. In contrast, MR-based tests combine high-quality oracles with diverse inputs and leverage developer-written setups for triggering target programs, resulting in higher test adequacy.

4.5 Related Work

4.5.1 Automated Identification of MRs.

Identification of proper MRs is a key step in applying MT to specific SUTs. To efficiently identify MRs, many automated approaches have been proposed. Earlier approaches identify MRs based on a set of predefined patterns [14, 23]. Zhang *et al.* [17] and Zhang *et al.* [18] proposed search-based approaches to inferring MRs. Tsigkanos *et al.* [101] proposed to use LLMs to identify variable relation and input transformation in scientific software. These approaches mainly synthesize MRs for specific domains. Shin *et al.* [102] proposed an approach to generating executable MRs from requirements specifications using LLMs, but it still requires human effort to implement supportive functions. Ayerdi et al. [19, 103] extend Terragni et al.'s work [28] to generate MRs via genetic programming. Nolasco *et al.* [104] proposed MemoRIA to infer equivalence MRs between methods and method sequences. Recently, Xu *et al.* [3] explored a new source to automatically derive MRs. They synthesize MRs from existing test cases where domain knowledge is embedded. This served as an effective approach to reusing many encoded MRs. Such encoded MRs are prevalent, but over 70% lack an input transformation function to support reusing them on more source inputs.

To reuse these invaluable MRs, this work proposes MR-Adopt to generate input transformation functions for such MRs. Integrated with the input transformations, these MRs are found helpful in enhancing test adequacy in our evaluation.

4.5.2 LLMs for Test Generation.

Researchers explored various LLM usages for test generation. Yuan *et al.* [82] studied the performance and limitations of ChatGPT in unit test generation. Xia *et al.* [105] built a fuzzer using LLMs as a generator of realistic test inputs and an engine for mutation. Tang *et al.* [106] compared the effectiveness of ChatGPT and Evosuite in unit test generation. Lemieux *et al.* [107] and Yang *et al.* [108] tried to promote the coverage of the tests generated by LLMs.

Different from these works, MR-Adopt does not use LLMs to generate tests directly. Instead, it generates the input transformation for the encoded MRs and reuses such MRs to enable more tests. In fact, using LLMs to generate correct and effective oracles and produce a large number of tests is found challenging [82]. In comparison, MR-Adopt reuses the human-written oracles in the encoded MRs, which are generally more reliable than LLM-generated oracles. Besides, MRs can be integrated with test input generation tools to produce abundant tests.

4.5.3 Enhancing LLMs for Code Generation.

LLMs are found powerful in code generation [79, 97], attracting numerous efforts to enhance the coding ability further. Some researchers designed more effective strategies of pre-training [109–111] and fine-tuning [112, 113]. Researchers also prompted LLMs with compilation messages to guide them to revise the generated code [82, 114, 115] or built a coding agent [116] to enhance LLM's code generation ability. In light of prompting with analogical reasoning [77], our work guides LLMs to generate more examples, identify the intention, and finally generate an input transformation matching the intention. Also, different from the approaches that rely purely on LLMs, MR-Adopt enhances the generated input transformation's quality by performing data-flow analysis to exclude irrelevant code segments from LLMs' responses and ranking the generated transformation functions based on validation with the output relation.

4.6 Chapter Conclusion

This work presents MR-Adopt, an LLM-based approach to generate input transformations for MRs encoded in test cases that lack explicit input relations. MR-Adopt allows these encoded MRs to be reused with new source inputs, enabling the generation of new tests and achieving higher test adequacy.

Experimental results show that MR-Adopt can generate effective input transformations, where 72% input transformations are generalizable to all prepared source inputs. When integrated with these transformations and new test inputs, encoded MRs increase line coverage by 10.62% and mutation score by 18.91%, demonstrating the practical usefulness of MR-Adopt 's transformations in enhancing test adequacy.

Data Availability. We have released the code of MR-Adopt and the experimental data at MR-Adopt's website [117].

CHAPTER 5

MR-COUPLER: AUTOMATED METAMORPHIC TEST GENERATION VIA FUNCTIONAL COUPLING ANALYSIS

Adopting MT is challenging. A key bottleneck is the construction of effective MRs [1], which requires domain-specific knowledge. Although several attempts have been made to explore the generation of MRs, these approaches suffer from (i) reliance on manual effort [12, 62, 118], (ii) assumptions of regression testing scenarios [19, 20], (iii) restriction to specific domains (e.g., autonomous driving) [14, 17, 18, 23, 118], or (iv) requirements for high-quality specifications [3, 16, 102]. All these studies rely on knowledge that is hard to obtain. Although a recent study [3] reported the possibility of mining the fragmented knowledge required by MRs from test cases, it also found such test cases to be rarely available: they account for only 1% of the studied test cases and are scattered in only 20% of the studied projects. The lack of automatic methodologies for constructing metamorphic test cases (MTCs) hinders the widespread adoption of MT. To ease its adoption, a technique without the above-mentioned limitations is expected. To construct such a technique, *a central challenge is to formulate MRs without relying on knowledge that is hard to obtain*.

Fortunately, it is observed that the *functional coupling between methods, which is readily available in the code, can be formulated as MRs*. For example, the pair of functions `encrypt` and `decrypt` can formulate an MR $x = decrypt(encrypt(x))$, as shown in Listing 13. This motivates us to formulate MRs by identifying such coupled method pairs. This idea offers several advantages: (i) *readily-available knowledge*: it relies solely on a pair of methods and their implementation, which is by construction available in the scenario of unit testing, (ii) *more tractable problem*: this transforms the challenging problem of deriving MRs into code understanding and relation reasoning, which can be effectively handled by current state-of-the-art large language models (LLMs) [79, 80, 82, 106]. For instance, although it is challenging to come up with MRs for a target method `encrypt`, when paired with a coupled method `decrypt`, it becomes easier for LLMs to understand their functionalities separately, realize that they are inverse functions, and then formulate a relation $x = decrypt(encrypt(x))$. (iii) *easier bug manifestation*: certain bugs can be revealed more easily with coupled computations. For instance, while it is difficult to reveal the bug in Listing 14 by calling `encrypt` and `decrypt` separately, it becomes easier with the MR $x = decrypt(encrypt(x))$. In summary, leveraging functionally coupled methods as a foundation for MR construction provides a practical and effective pathway to automate metamorphic testing and broaden its applicability.

However, leveraging functionally coupled methods to construct MRs requires addressing two technical issues. First, given a target method, there can be dozens of candidate method pairs, and it is expensive to enumerate all possible method pairs blindly for MR construction. Thus, there is a need for a precise mechanism to identify functionally coupled method pairs, which provides better focal methods for subsequent MR construction. Second, while LLMs enable MTC generation via code understanding and reasoning, the resulting MTCs can be invalid due to hallucination [119]. Therefore, an effective mechanism is needed to validate the generated MTCs, which allows us to avoid overwhelming developers with false alarms [28].

To tackle these technical issues and effectively generate MTCs, this work proposes MR-Coupler, an automatic MTC generator for a given target method. It operates in three phases. First, it identifies functionally coupled methods as ingredients for MR construction, based on their signatures and implementations. This addresses the first technical issue, based on our observation that developers often write MTCs for methods that operate on the same data structures or share common dependencies (e.g., APIs and class fields). Next, it employs LLMs to generate MTCs based on each identified functionally coupled method pair by providing relevant and minimal context. Specifically, MR-Coupler instructs LLMs to understand their functionalities and reason about potential MRs between them. To reduce hallucinations that lead to invalid code, MR-Coupler provides examples of MTCs and retrieve API usages for LLMs to follow. Finally, it validates the generated candidate MTCs via test amplification and mutation analysis. To validate the MTCs without a given ground truth, MR-Coupler creates mutants from the original program by injecting artificial faults, and expect more amplified MTCs (from the candidate MTC) to pass on the original version compared with the faulty mutants. This filtering strategy is based on a property of MT: the MR embedded in a correct MTC should apply to many other inputs to effectively kill mutants [3].

This work evaluated MR-Coupler on (i) 100 human-written MTCs with corresponding target methods and (ii) 50 real-world bugs as tasks for evaluation. MR-Coupler successfully generates valid MTCs for over 90% of tasks, achieving a 64.90% improvement in valid MTC generation and a 36.56% reduction in false alarms compared with baselines. The MTCs generated by MR-Coupler found 44% of the 50 real bugs. The key components play crucial roles in MR-Coupler:

Listing 13 An Example of MTC that Encodes the MR $x = decrypt(encrypt(x))$ over `encryptText` and `decryptText`

```java
@Test
public void testEncryptDecrypt() throws Exception {
    String plainText = "Hello AES!";
    SecretKey secKey = AESEncryption.getSecretEncryptionKey();
    // Encrypt the plaintext: invoke on the source input, and produce the source output
    byte[] cipherText = AESEncryption.encryptText(plainText, secKey);
    // Decrypt the ciphertext: invoke on the follow-up input, and produce the follow-up output
    String decryptedText = AESEncryption.decryptText(cipherText, secKey);
    assertEquals(plainText, decryptedText); // output relation assertion
}
```

Listing 14 Code of Methods `encryptText` and `decryptText`

```java
public static byte[] encryptText(String plainText, SecretKey secKey) {
    Cipher aesCipher = Cipher.getInstance("AES");
    // aesCipher.init(Cipher.ENCRYPT_MODE, AESEncryption.defaultKey);   // bug
    aesCipher.init(Cipher.ENCRYPT_MODE, secKey);                        // fix
    return aesCipher.doFinal(plainText);
}
public static String decryptText(byte[] byteCipherText, SecretKey secKey) {
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.init(Cipher.DECRYPT_MODE, secKey);
    return aesCipher.doFinal(byteCipherText);
}
```

this work found that functional relevance between methods guides LLMs to produce valid, bug-revealing MTCs, while MTC amplification and validation steps further improve bug detection and halve false alarms. Last but not least, MR-Coupler can mimic developers in using functionally coupled methods and achieves over 90% MR-skeleton consistency with human-written MTCs, highlighting its potential to assist developers in constructing MTCs.

In summary, this work makes the following contributions.

- To the best of the authors' knowledge, this work is the first to construct MRs by leveraging the functional coupling between methods. This work's approach relies solely on the code under test, and thus enables easier MR construction and lowers the barrier to adopting MT.
- This work designed and implemented MR-Coupler, an automatic approach to generate concrete metamorphic test cases. MR-Coupler instructs LLMs with relevant and minimal context for MR construction, and validates the generated MTCs with a novel filtering mechanism based on mutation analysis.
- This work conducted extensive experiments to evaluate the effectiveness of MR-Coupler, including the validity of generated MTCs, the capability of revealing real bugs, and the similarity of generated MTCs to human-written MTCs.
- This work made MR-Coupler and our experimental data publicly available to facilitate future research. The artifact is available at the website of MR-Coupler [117].

!TeX root = ../../../main.tex

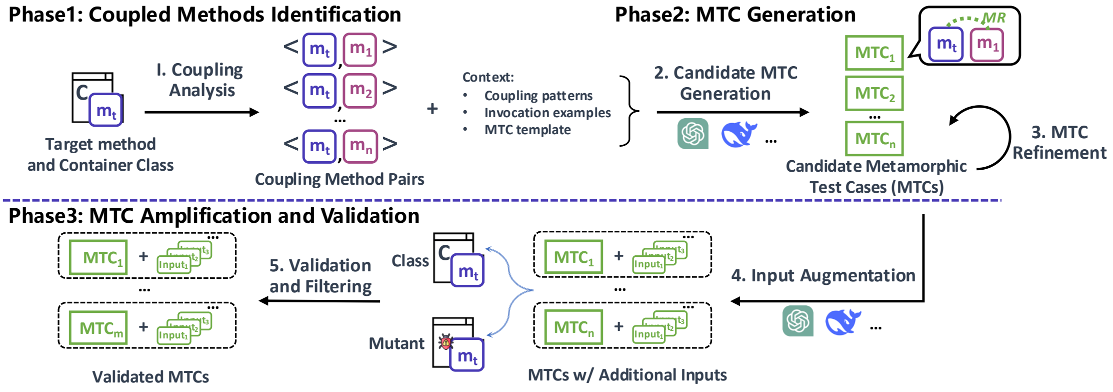


Figure 5.1: An overview of MR-Coupler

5.1 Approach

This section presents MR-Coupler, an automated MTC generator based on functional coupling. Figure 5.1 presents an overview of MR-Coupler. Given a target method and its container class as input, MR-Coupler produces a set of MTCs. Specifically, the generation process consists of three phases:

1. *Coupled Methods Identification*. In the first phase, MR-Coupler identifies functionally coupled methods that will be paired with the target method for MR construction. These methods are relevant to the target method in their intention or functional behavior. Such relevance can lead to potential MRs over their functionalities.
2. *MTC Generation*. In the second phase, MR-Coupler leverages LLMs to generate MTCs for the method pairs yielded by the first phase. To mitigate the impact of LLM hallucination [119], MR-Coupler provides example usage of the involved methods and a template of MTC in the prompt. MR-Coupler also performs subsequent refinement based on the execution output of the generated MTCs.
3. *MTC Amplification and Validation*. The third phase amplifies the MTC generated by the second phase with additional inputs. This phase serves two goals. First, this serves as a validation for the MTC based on a necessary property: a valid MTC should not have a lower pass rate on the original version than on the mutants. Second, the amplified MTCs can help reveal more bugs by exercising a broader range of program behaviors.

5.1.1 Phase 1: Coupled Methods Identification

This phase is to identify the functionally coupled methods to combine with the target method as method pairs for MR construction. However, given a target method and its container class, it is non-trivial to identify the functionally coupled methods that are suitable for MR construction.

On the one hand, classes often contain dozens of methods. For example, in our dataset, there are over 30 candidate methods in a class for each target method on average. Naively including all candidates not only increases the cost of LLM tokens but also risks of introducing noisy context, which distracts LLMs. The difficulty is to identify those methods with relevance (e.g., producer–consumer, equivalent or inverse functions) suitable for MR construction.

Characterization of Method Coupling. Although the number of possible method combinations is large, the author found the method pairs used for MR construction often exhibit certain patterns in their relation: sharing relevant intentions, similar implementation behavior, or state interference.

For example, for two methods that have the inverse relationship, methods perform opposite input and output type transformations (e.g., `encryptText:(String,SecretKey)->byte[]` and `decryptText:(byte[],SecretKey)->String`) [120]. For two methods that have the producer-consumer relationship, one method updates or produces states (e.g., fields of an object), and another accesses them [121]. For two methods that have an equivalent or similar functionality relationship, e.g., overloading methods, they invoke the same APIs, access or update the same fields of an object. For two methods that have the composition or specialization relationship, one method internally calls or extends the functionality of another [122]. More examples can be found in our artifact [117].

Despite the diversity in these relationships, most of them can be captured by three complementary patterns:

(i) *Relevant Intention*: The method pairs that are designed to perform related functionalities. This can be captured from their signatures, including method name, parameter types, and return type, which indicate the data types they consume and produce.

(ii) *Similar Implementation Behavior*: The method pairs that have similar behaviors in their implementations. This can be captured by analyzing their function calls, indicating that they perform comparable or related operations.

(iii) *Potential State Interference*: The method pairs where the invocation of one method can affect the behaviors of the other. This can be reflected by the class fields and object states that they access or modify.

Coupling Analysis. This step aims to identify functionally coupled methods characterized by the above three patterns.

Relevant Intention: Methods with relevant intention can often lead to potential MRs (e.g., methods `encryptText` and `decryptText`). To identify such method pairs, MR-Coupler analyzes the method name, parameters, and return types. Specifically, given a target method m_t, a candidate method m_i is considered to have relevant intention if: (i) m_t and m_i share the same method name (indicating an overloading relationship of similar purpose), or (ii) m_t and m_i share common name tokens and operate on the same parameter or return types, which increases the likelihood that they manipulate the same data structures for relevant functionalities. These heuristics allow MR-Coupler to identify methods with relevant intentions (e.g., inverse methods, overloading methods, etc.).

Similar Implementation Behavior: Methods with relevant functionalities may not always come with similar names or signatures. Their relevance can manifest at the behavioral level. Therefore, to capture such relevance, MR-Coupler extracts the set of functions (e.g., APIs) invoked within each method. It then analyzes three types of invocation relationships: (i) whether m_t directly invokes m_i (or vice versa), indicating relationships like specialization or composition, (ii) whether m_t and m_i share common invoked APIs, suggesting similar behavior. This allows MR-Coupler to identify methods with similar implementation behavior.

Potential State Interference: Beyond invocation and API usage, method pairs can lead to potential MRs if one method can affect the behavior of the other. Such a relationship is often reflected by the field access and updates. Therefore, MR-Coupler analyzes the fields accessed and updated by each method. Given a target method m_t, it considers m_i relevant if: (i) m_i updates fields that m_t later reads (or vice versa), indicating a data-flow dependency and relationship like producer-consumer, (ii) m_i and m_t access the same fields, or (iii) m_i and m_t update the same fields, suggesting similar behavior. This allows MR-Coupler to identify potential state interference between methods.

Given a target method, MR-Coupler analyzes all the methods within its container class, and yields the method pairs that match any of the patterns. Note that a method pair may satisfy multiple patterns.

5.1.2 Phase 2: MTC Generation

Given the set of coupling method pairs, this phase employs LLMs to come up with MRs and generate concrete MTCs that conduct MT. However, generating valid MTCs remains challenging even with state-of-the-art LLMs [123, 124]. Many methods require complex object instantiations, parameter configurations, or specific environmental setups, making valid method invocation difficult. Without concrete usage examples, LLMs are prone to hallucinations, often producing code that references non-existent classes, APIs, or fields. Moreover, generated test cases must conform to the steps of MT (i.e., constructing source and follow-up inputs, invoking methods, and asserting output relation)

To address these challenges, MR-Coupler firstly provides LLMs with contextual guidance (e.g., method invocation examples and MTC template). After prompting the LLMs to generate MTCs, MR-Coupler further refines them based on the execution output.

Candidate MTC Generation. The details of this step are as follows.

Invocation example preparation: To help LLMs construct valid method invocations, MR-Coupler retrieves method invocation examples from the project under test and uses them as part of the contextual guidance.

Specifically, for each method pair $\langle m_t, m_i \rangle$, MR-Coupler searches for invocations of any of the two methods in the test code. MR-Coupler scans the test files (under the `/test/` directory) in the project under test and uses `JavaParser` [88] to identify test methods annotated with `@Test`. It then checks whether each test invokes either m_t or m_i. If so, the test is collected as an invocation example. All retrieved examples are aggregated, with at most three examples retained for each method, and provided to the LLMs as contextual guidance, increasing the likelihood that generated tests correctly instantiate objects and invoke methods.

Prompt Design: Listing 15 shows a simplified prompt template used by MR-Coupler for MTC generation. Referring to the prompt design in recent studies [80, 82, 83], the prompt includes: (i) a system message specifying the role of LLM and its tasks, (ii) the code of the paired methods, (iii) the identified coupling patterns on the paired method (iv) invocation examples, (v) the skeleton of the container class (fields and method signatures), and (vi) a MTC template that specifies the required deliverable. This structured prompt provides both contextual information (ii–v) and task description (i and vi), guiding the LLM to generate syntactically correct MTC. The details of employed LLMs and their configuration can be found in Section 5.2.2. The output of this step is a set of candidate MTCs, each implemented as a standalone test class.

MTC Refinement. Consistent with prior observations [80, 87], LLM-generated code frequently fails to execute commonly due to errors such as ``` ``cannot find symbol'' ```. These errors typically arise from two sources: (i) referencing non-existent classes, APIs, or fields (hallucinations), or (ii) missing dependencies (e.g., absent import statements).

To deal with this issue, MR-Coupler refines each non-compilable or non-executable MTC. First, the error message is provided back to the LLM to request an automatically revised version.

Listing 15 Prompt Template for MTC Generation

~~~
You are an expert in Java programming and metamorphic testing, your task is to: ...

# Code of the paired method
    ```java
      byte[] encryptText(String plainText, SecretKey secKey) {
        ...
      String decryptText(byte[] byteCipherText, SecretKey secKey) {
        ...
    ~~~

# Coupling patterns on the paired methods
### Relevant Intention:
* `encryptText` and `decryptText` operate on the same set of parameters and return types,
 ↪ but with different transformations.
    * `encryptText`: (String, SecretKey) -> byte[] , `decryptText`: (byte[],SecretKey) ->
     ↪ String
### Similar Implementation Behavior:
* both `encryptText` and `decryptText` invoke same APIs: `Cipher.getInstance(``AES'')`
  ...
# Invocation examples
  ...
# Skeleton of the container class
  ...
# Deliverable
    ```java
    public class $testClass${
        @Test
        public void $testCase$() {
          <MTC Template>
          ...
    ```
```

If the revised MTC still fails to execute, MR-Coupler tries to fix the missing dependencies issue by statically analyzing the code using `JavaParser` to extract unresolved class names and searches for potential classes defined or imported in the project under test, and then adds the necessary import statements. Finally, this phase results in a refined set of MTCs, which are subsequently used for amplification and validation.

5.1.3 Phase 3: MTC Amplification and Validation

This phase aims to validate the candidate MTC generated in the previous phase and diversify the test inputs. Specifically, MR-Coupler amplifies the MTC with additional inputs, and leverage a property of MT to refute invalid MTCs. This is because the previous phase can generate MTCs that encode *invalid* metamorphic relations (MRs). To refute such MTCs, MR-Coupler opted for mutation analysis: MR-Coupler creates mutants from the original program by injecting artificial faults, and expect more amplified MTCs (from the candidate MTC) to pass on the original version compared with the faulty mutants. This filtering strategy is based on a property
```

**Listing 16** Simplified example of an MTC with $M$ additional inputs

~~~
```java
public class AESEncryptionTest{
    @Test
    public void testEncryptDecrypt_input1() {
      String text = "Hello!"; SecretKey key = AESEncryption.getSecretEncryptionKey();
      byte[] encryptedText = AESEncryption.encryptText(text, key);
      String decryptedText = AESEncryption.decryptText(encryptedText, key);
      assertEquals(text, decryptedText);
    }
    @Test
    public void testEncryptDecrypt_input2() {
      String text = NULL; SecretKey key = NULL;
      ...
    public void testEncryptDecrypt_input3() {
      String text = "~!@"; SecretKey key = AESEncryption.getSecretEncryptionKey();
      ...
    public void testEncryptDecrypt_input4() {
      String text = "_1234567890"; SecretKey key = AESEncryption.getSecretEncryptionKey();
      ...
    public void testEncryptDecrypt_inputM() {
      String text = ""; SecretKey key = AESEncryption.defaultKey;
      ...
}
```
~~~

of MT: the MR embedded in a correct MTC should apply to many other inputs to effectively kill mutants [3]. In addition, as a side product, the amplified MTCs can also help exercise a broader range of program behaviors and increase their bug-revealing capability.

Input Augmentation. MR-Coupler amplifies each MTC by generating additional inputs to its MR to exercise a broader range of program behaviors, thereby enhancing its bug-revealing capability. To generate these inputs, MR-Coupler employs LLMs by appending new instructions to the conversation for MTC generation and prompts the model to: (i) review the previous conversation and context, (ii) review the previously generated MTC, (iii) apply its MR to new inputs by replacing the original input with ⟨M⟩ new inputs (such as boundary values, random data, or special characters) in the form of new test cases ($M = 10$ by default), and (iv) output the new test cases within the same class following the naming convention (`testMTC_newInput1()`, …, `testMTC_newInputM()`). Listing 16 shows a simplified example of amplified MTC with 5 new inputs.

Validation and Filtering. Given an amplified MTC with additional inputs, MR-Coupler executes it on both the *original* version of the target program, and a *mutated* version produced by injecting faults using Major [125].

For each MTC, MR-Coupler computes the pass rate p on the original version and p' on the mutated version. The validation property requires that $p > p'$: a valid MTC should pass consistently on a correct implementation and fail on a buggy implementation. If this property is

Listing 17 Example of `encryptText` and `encryptTextWithAbecedarium` with bugs

```
public static byte[] encryptTextWithAbecedarium(String plainText,SecretKey secKey,String
 ↪  abecedarium)
{
    Cipher aesCipher = Cipher.getInstance("AES");
    aesCipher.abecedarium = AESEncryption.abecedarium;        // bug
    // aesCipher.abecedarium = abecedarium;                   // fix
    aesCipher.init(Cipher.ENCRYPT_MODE, secKey);
    return aesCipher.doFinal(plainText);
}
```

violated ($p < p'$), the MTC is flagged as invalid and discarded. For instance, consider a mutant (`encryptText` uses wrong `key` shown in Listing 14) and a valid MR $decrypt(encrypt(x)) = x$ with additional inputs in Listing 16. Most inputs pass on the original version ($p = 80\%$, except when `text=NULL` throws an illegal input exception) but fail on the mutant ($p' = 20\%$, since only the case where `key = AESEncryption.defaultKey` coincidentally matches the default key succeeds). Because $p > p'$, this MTC is retained as valid. By contrast, an LLM-generated invalid MRs $encrypt(plainText, secKey) = encryptTextWithAbecedarium\ (plainText, secKey, abecedarium)$ is observed. When tested against a mutant where `encryptTextWith Abecedarium` fails to set up the `abecedarium` (Listing 17), most inputs in Listing 16 fail on the original version ($p = 20\%$, except when the user-defined `abecedarium` happens to match the default). On the mutant, all inputs pass ($p' = 100\%$) because the `abecedarium` is ignored entirely. Since $p < p'$, this MTC is classified as invalid and filtered out.

When $p = p' = 100\%$, two interpretations are possible: (i) the injected mutants are ineffective and do not affect the tested behavior, or (ii) the MTC is ineffective in exposing mutants. In such cases, MR-Coupler conservatively retains the MTC. Since $p \geq p'$ is a necessary condition for a valid oracle, increasing the number of additional inputs raises the likelihood of exercising diverse program behaviors and triggering differences between the original and mutated versions, thereby improving the effectiveness of MR-Coupler's validation.

After validating each generated MTC, MR-Coupler finally outputs validated MTCs.

!TeX root = ../../../main.tex

5.2 Evaluation

This section presents our evaluation of MR-Coupler. Specifically, this work aims to answer the following research questions (RQs).

RQ9 Validity: *How effective is* MR-Coupler *at generating MTCs?* This RQ investigates the overall effectiveness of MR-Coupler. Specifically, this work assesses MR-Coupler regarding the validity of the generated MTCs, i.e., whether they are syntactically correct, entail necessary steps of MT, and do not produce false alarms. In addition, this work compares MR-Coupler with vanilla-LLM-based baselines to understand the superiority of our approach.

RQ10 **Bug-Revealing Capability:** *How effective is* MR-Coupler *in revealing real-word bugs that discovered by human-written MTCs?* This RQ aims to understand the effectiveness of MR-Coupler in revealing bugs in practical scenarios. Compared to seeded bugs (e.g., mutants), real-world bugs are often more sophisticated. Thus, this work evaluates whether the MTCs generated by MR-Coupler can detect real-world bugs as the human-written MTCs do in the same test subjects.

RQ11 **Abalation Study:** *How does each step contribute to the effectiveness of* MR-Coupler*?* MR-Coupler incorporates three key steps: *coupling analysis* to identify functionally related methods, *input augmentation* to amplify generated MTCs, and *mutation analysis* to validate MTCs. This RQ performs an ablation study to understand how each of these steps contributes to the overall effectiveness of MR-Coupler in generating valid and bug-revealing MTCs.

RQ12 **Similarity:** *Do the MTCs generated by* MR-Coupler *share the same MR skeletons as human-written ones?* This RQ evaluates whether the MTCs generated by MR-Coupler can mimic developers' practices in selecting functionally coupled method pairs and constructing input and output relations. This demonstrates the potential of MR-Coupler to assist developers in MTC construction, facilitating developers in integrating the generated tests into their codebase and easing subsequent maintenance.

5.2.1 Datasets

This work prepared two datasets to answer the four RQs. The first dataset includes pairs of target methods together with corresponding human-written MTCs available in open-source projects to evaluate the validity and similarity of the generated MTCs (RQ9 and RQ12). The other is a subset from the first dataset, including only the cases whose MTCs can reveal bugs on a historical buggy version of the target program, for evaluating the bug-revealing capability of generated MTCs (RQ10).

Human-Written MTCs. The first dataset contains 1,471 MTCs written by developers in open-source Java projects. Each entry in this dataset consists of a human-written MTC and a corresponding pair of MR-coupled methods. These MTCs are valid and executable. Such a dataset is leveraged to (i) evaluate the validity of automatically generated MTCs (RQ9), by running them on executable target methods, and (ii) measure the similarity between the human-written MTCs and MR-Coupler generated MTCs, by checking whether they encode the same MR-skeletons (RQ12).

To construct such a dataset, this work adopted a strategy similar to Xu et al. [3]. Specifically, this work collected a list of high-quality Java projects (i.e., with at least 50 stars) from GitHub. The query was done on December-16, 2024, which returned over 24,000 projects. This work then ran MR-Scout [3] to discover human-written MTCs from these projects, which yielded 46,006 candidate MTCs. With these candidates, this work applied three filtering criteria to select the valid and executable MTCs:

(i) they must compile, as this work needs to compile and run the test cases in our experiments;
(ii) they must pass in the latest version of the project to ensure the MTCs are valid; and
(iii) the commit introducing these MTCs must mention an issue number in its commit message (e.g., containing "`#123`"). This work prioritizes such tests since they are often extensively discussed and reviewed to disclose the issues and thus tend to be of high quality.

The whole processes yielded a dataset containing 1,471 entries. Each entry in this dataset consists of a human-written MTC and a corresponding pair of MR-coupled methods. The author ran MR-Scout [3] to obtain the corresponding pair of MR-coupled methods for each MTC. For example, in the MTC that encodes the relation $x = decrypt(encrypt(x))$ (Listing 13), `encryptText` and `decryptText` are the MR-coupled methods and will be identified by MR-Scout. This work uses the first invoked method `encryptText` as the target method and take `decryptText` as the ground truth of a coupled method. Each entry formulates an MTC generation task used for our experiments of RQ9 and RQ12.

Bug-revealing MTCs. The other dataset is made up of 50 entities with bug-revealing MTCs filtered from the first dataset. These entities are used to evaluate whether MR-Coupler can generate effective MTCs to reveal real bugs as the human-written MTCs do (RQ10). Such entities are identified from the first dataset by checking whether their MTC will fail on a buggy version while pass on a fixed version. Specifically, an issue report may be resolved through commits; thus, for each issue-associated MTC, the author identifies two versions of the project: (i) the potential *buggy version*, defined as the commit before all issue-related commits; (ii) the potential *fixed version*, defined as the last commit of all issue-related commits. The author then executes the MTC on both versions. This work considers an MTC bug-revealing only if it fails on the buggy version and passes on the fixed version.

This process required significant manual effort to set up specific project environments for multiple versions of each project and resolve complicated dependency issues. This work ultimately reproduced 50 MTC-bug pairs, obtaining their corresponding buggy and fixed program versions, which form the benchmark for evaluating the bug-revealing capability of MR-Coupler (RQ10).

5.2.2 Evaluation Setup

In this section, this work presents the evaluation setup. This work introduces the LLMs used, baselines, and the experiment environment.

Employed Large Language Models.

MR-Coupler employs LLMs to generate MTCs and their alternative inputs. In the evaluation, this work includes representative state-of-the-art LLMs [93], covering general-purpose, coding, and reasoning LLMs from well-known model families. Specifically, they are *GPT-4o mini* from OpenAI [126],*Qwen3-coder-Flash* from Alibaba [92], *DeepSeek-V3.1* and *DeepSeek-V3.1-Think* from DeepSeek [91]. Following a typical setup in recent studies [76, 80, 85], for each MTC generation task, this work repeated the generation process five times with a temperature setting of 0.2.

Baselines. To the best of our knowledge, there is no existing fully automated and domain-agnostic approach to generate metamorphic test cases for a given program under test. Although some approaches are proposed to generate domain-specific MRs [18, 118], or synthesize MRs based on human-prepared materials [3, 80, 104] or manual effort [102] (discussed in Section 5.3), adapting them into comparable automated domain-agnostic baselines is non-trivial. Given the proven effectiveness of LLMs in code [79, 80, 97] and test generation [82, 106], this work sets *directly prompting LLMs* as a baseline. In this baseline, this work allows LLMs to conduct a round of revision to the generated code based on the execution feedback as in our method, which is found to be an effective common post-processing to enhance code generation [82]. The baseline uses a similar prompt template (Listing 15), and follows the same refinement step in Section 5.1.2.

Experimental results show that a target method can be paired with 6.93 relevant methods (rounded to 7) by MR-Coupler on average. Therefore, to have a fair comparison, this work instructed the baseline LLMs to generate 8 candidate MTCs, which correspond to the target method itself, plus the 7 additional relevant methods. For each task, this work repeated the generation process five times with a temperature setting of 0.2, consistent with MR-Coupler's configuration.

Experimental Environment. All experiments were conducted on a machine with a 64-core AMD Ryzen Threadripper PRO 3995WX CPU and 512 GB RAM. The LLMs in our evaluation are running on cloud platforms and accessed via the official APIs of OpenAI, Alibaba, and DeepSeek.

5.2.3 RQ9: Validity of Generated MTCs

RQ9 aims to evaluate the overall effectiveness of MR-Coupler in generating valid MTCs. To this end, MR-Coupler is evaluated on the target methods in the dataset of human-written MTCs. This work also compares it against the baselines (Section 5.2.2).

Experiment Setup. Experiments were conducted at scale using four LLMs, with each MTC generation task repeated five times (Section 5.2.2). Running MR-Coupler and the baselines on all 1,471 tasks in the human-written MTC dataset is time-consuming and unaffordable. Therefore, this work runs MR-Coupler and baseline approaches on 100 randomly sampled entries in the dataset. Such a sample size ensures a confidence level of 95% and a margin of error ≤ 10%.

For each task (i.e., target method), MR-Coupler generates multiple MTCs. This work presents the total number of generated test cases, and measure the effectiveness using the following metrics:

- *Percentage of Executable MTC:* The proportion of executable MTCs to all generated test cases, where an MTC is executable if it (i) compiles and runs without errors, (ii) satisfies the necessary properties of an MTC. Following the definition from Xu *et al.* [3], an MTC must meet two properties: (*P1*) it must contain at least two method invocations with two inputs separately, and (*P2*) it must contain one assertion checking the relation between the inputs and outputs of the method invocations in *P1*. This work re-ran their tool to automatically verify each generated test case.
- *Percentage of Valid MTC:* The proportion of valid MTCs to all generated test cases, where a valid MTC is an executable MTC that passes on the latest project version. This work assumes the latest version of a target method is of low probability to be buggy, as it has passed human-written MTCs.
- *Number of Successful Tasks:* The number of target methods for which at least one valid MTC is generated.
- *Percentage of False Alarm:* The proportion of invalid MTCs to all executable MTCs, where an invalid MTC satisfies the properties but fails on the latest version.

Table 5.1: Effectiveness of MR-Coupler in Generating Valid MTCs for 100 Target Methods

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		**Improv.**
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
Num. of Generated TCs	3923	4176	3984	3911	3968	3626	3971	4151	-
Pct. of Executable MTC	50.40%	83.69%	60.02%	92.66%	60.26%	93.08%	62.08%	88.44%	**54.35%**
Pct. of Valid MTC	40.66%	71.84%	47.52%	80.98%	52.60%	84.94%	55.35%	83.57%	**64.90%**
Num. of Successful Tasks	62	92	76	91	82	95	85	98	**24.82%**
Pct. of False Alarm	19.32%	14.16%	20.70%	12.61%	12.71%	8.74%	10.83%	5.50%	**36.56%**

Experiment Results. Table 5.1 shows the result of MR-Coupler in generating valid MTCs for 100 target methods. MR-Coupler successfully generated valid MTCs for over 90 target methods, with its best performance achieved when using *DeepSeek-V3.1-Think*. Specifically, with *DeepSeek-V3.1-Think*, MR-Coupler produced valid MTCs for 98 of 100 target methods, with only 5.5% of false alarms. Compared to the baseline, MR-Coupler achieves a 64.90% higher valid MTC percentage with a 36.56% fewer false alarms. Even with the weakest model (*GPT-4o mini*), MR-Coupler still outperformed a baseline with a much more powerful model (i.e., *DeepSeek-V3.1-Think*) in generating valid MTCs.

The improvement in valid MTC generation can be attributed to two key factors. On the one hand, providing LLMs with functionally coupled methods serves as a hint, effectively inspiring them to infer valid MRs and then generate valid MTCs, thereby reducing hallucinations. Without such context, coming up with MRs from scratch is challenging for LLMs, leading to higher rates of invalid MRs. On average, MR-Coupler identified 6.93 relevant methods per target method from 30.44 candidate methods in their container classes, significantly narrowing the enumeration space. On the other hand, retrieving real invocation examples helps LLMs construct valid input objects and correctly invoke methods, particularly when methods require complex object instantiations.

For example, the target method `estimateCNF` [127] requires a less common `CNFEstimation` object as input. Without a concrete example, LLMs frequently failed to construct the object correctly and even hallucinated non-existent APIs such as `getEstimator()`. This context helps generate 83.69% to 93.08% executable MTCs, which are 54.35% more compared with baselines.

Failure analysis. Even built with *DeepSeek-V3.1-Think*, MR-Coupler fails to generate any valid MTCs for two target methods and still exhibits a 5.5% false-alarm rate. The main reasons are as follows: (i) Some target methods require access to external or environmental resources (e.g., a JSON file or environment variable), but no invocation examples were available in the repository as the context. As a result, MR-Coupler fails to configure these resources in the generated MTCs, leading to non-executable tests. (ii) MR-Coupler relies on Major [125] to

Table 5.2: Bug-Revealing Results of MR-Coupler on 50 Bugs

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		**Improv.**
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
Num. of Generated TCs	1987	2740	1976	2086	2024	1797	2007	2661	-
Pct. of Bug-revealing MTCs	3.84%	6.53%	4.14%	7.29%	5.21%	6.60%	3.92%	7.77%	**67.65%**
Num. of Revealed Bugs	4	15	5	20	7	16	7	22	**229.46%**

generate mutants for validation (Section 5.1.3). For some target methods, Major failed to execute due to environmental issues (e.g., uncompilable dependencies), preventing mutant generation and disabling the validation step that filters false alarms. (iii) In some cases, the generated MTCs cannot reveal the injected mutants, and do not expose behavioral differences between the base and mutated versions – the pass rates are identical, causing MR-Coupler to retain false alarms.

Answer to RQ9: MR-Coupler successfully generates valid MTCs for over 90% of tasks with, achieving 64.90% and 36.56% improvements in generating valid MTCs and reducing false alarms, respectively, compared with baselines.

5.2.4 RQ10: Bug-revealing capability

Experiment Setup. This RQ evaluates the capability of MR-Coupler in revealing real bugs, especially for those originally discovered by metamorphic test cases. With the collected 50 MTC-bug pairs (Section 5.2.1), for each bug, this work takes the buggy method as the target method. This work measures the bug-revealing capability by the following two metrics:

- *Percentage of Bug-Revealing MTCs:* The proportion of generated MTCs that are bug-revealing, where a bug-revealing MTC is defined as an executable MTC that fails on the buggy version but passes on the fixed version of the target method.
- *Number of Revealed Bugs:* The number of bugs for which at least one generated MTC fails on the buggy version and passes on the fixed version.

Experiment Result. As shown in Table 5.2, MR-Coupler (*DeepSeek-V3.1-Think*) performed the best, successfully revealing 22 real-world bugs with a bug-revealing MTC percentage of 6.53%. Specifically, it revealed 15 more bugs and achieved a 98.21% improvement in bug-revealing MTC percentage compared with the baseline. When combined with other models, MR-Coupler consistently outperforms baselines, revealing 9∼15 additional bugs and achieving an average 67.65% increase in bug-revealing MTC rate.

Based on manual inspection, the author attribute the improvement to two main factors: (i) *Relevance-aware MR construction.* Some bugs can only be triggered by the MRs that couple specific method pairs, and MR-Coupler can generate such MRs through its coupling analysis step. For example, a bug in the "producer" method `randomRepo` can be revealed when coupled with the "consumer" method `repos`, which accesses the coordinates of the newly created repository [128]. Both methods invoke the same API `MkRepo` and access the same fields (`storage` and `self`), allowing MR-Coupler to identify their relevance and generate an effective MR. (ii) *Input augmentation.* Some bugs require specific input to trigger. By applying additional inputs (Section 5.1.3), MR-Coupler exercises a broader range of program behaviors and uncovers such corner-case bugs. For example, a bug in `previousClearBit` manifests only when processing inputs near array boundaries (e.g., `int i=1<<16`) [129]. In addition, the performance comparison among the evaluated LLMs shows that *DeepSeek-V3.1-Think* revealed the highest number of bugs, likely due to its superior reasoning ability. By analyzing the code of target methods and understanding the relevance between methods, it can reason about potential fault-prone scenarios and construct MRs that expose them.

By analyzing the overlap of bugs revealed by MR-Coupler with different LLMs, it was observed that a total of 28 unique bugs were revealed. 8 of the 28 bugs were detected by all models, and both *DeepSeek-V3.1-Think* and *Qwen3-coder-Flash* uniquely revealed two additional bugs. This suggests that *combining models could further improve the bug-revealing capability*. This is an interesting strategy for future enhancement.

It was observed that 19 out of 22 bugs revealed by *DeepSeek-V3.1-Think* were detected by more than one distinct MTC. For example, a bug in a ``` ``multiply'' ``` method could be revealed by multiple MRs, such as $a = divide(multiply(a, b), b)$ or $multiply(subtract(a, b), c) = subtract(multiply(a, c), multiply(b, c))$ [130]. This highlights that *MR diversity plays a role in enhancing the bug-revealing capability.*

Failure analysis. While MR-Coupler successfully detected 22 (44%) real bugs originally revealed by human-written MTCs, it failed to expose some others. A major reason is that certain methods under test are highly domain-specific and implement complex business logic. In such cases, simply providing the code of the method is insufficient for an LLM to fully understand its functionality and intended specification. For example, in a bug related to "compaction file metrics" in Apache IoTDB [131], understanding the expected behavior of the `doCompaction` method requires deeper module-level or even project-level knowledge. Automatically extracting such background context and enabling an LLM to reason about domain-specific business logic remains a challenging and promising direction for future research. In other cases, constructing inputs required access to external resources, such as environment variables or specific file contents [132]. When no concrete examples were retrieved in the project under test, MR-

Coupler generated MTCs failed to set up that, resulting in non-executable or non-bug-revealing tests. Automatically and completely retrieving and adapting such project-level context for test generation is an open challenge for future work.

Answer to RQ10: MR-Coupler can successfully detect 22 (out of 50) real-world bugs originally discovered by human-written MTCs. However, some unrevealed bugs are rooted in domain-specific business logic, requiring model-level or even project-level context to construct bug-revealing MTCs. Effectively leveraging such context remains an open challenge for future work.

5.2.5 RQ11: Ablation Study on MR-Coupler

Experiment Setup. This RQ aims to evaluate the contribution of major steps in MR-Coupler to its overall effectiveness in generating valid and bug-revealing MTCs. This work uses the same tasks and metrics as in RQ9 (validity) and RQ10 (bug-revealing capability) This work created three ablated variants of MR-Coupler (v_1, v_2, and v_3) by ablating three steps to analyze their contribution. This work chose MR-Coupler built with *DeepSeek-V3.1-Think* which achieves the best result in RQ9 and RQ10 (Sections 5.2.3 and 5.2.4). The variants are as follows:

- v_1**: MR-Coupler w/o Coupling Analysis.** This variant disables the *Coupling Analysis* step (Section 5.1.1), meaning no functionally coupled methods and corresponding invocation examples are provided as context to LLMs during the MTC generation.
- v_2**: MR-Coupler w/o MTC Amplification.** This variant disables the *Input Agumentation* step (Section 5.1.3), thus no additional inputs are generated to amplify MTCs.
- v_3**: MR-Coupler w/o MTC Validation.** This variant disables the *Validation and Filtering* step (Section 5.1.3), meaning all generated MTCs are retained without filtering.

Experiment Result. As shown in Table 5.3, disabling *Coupling Analysis* (v_1) led to a 32.65% decrease in valid MTC rate and a 56.62% decrease in bug-revealing rate. This suggests that leveraging functional coupling as an explicit hint is crucial for generating valid MTCs and reducing hallucinations. This aligns with the findings in RQs of validation and bug-revealing capability.

When disabling *Input Agumentation* (v_2), the revealed bugs significantly decreased by 40.91% (from 22 to 13). This highlights that applying generated MRs to additional inputs strengthens generated MTCs by exercising a wider range of program behaviors. Nevertheless, even without input augmentation, MR-Coupler revealed more bugs compared with the baseline, indicating that MRs over multiple methods already contribute to bug revealing, and MTC amplification further boosts the bug-revealing rate by 89.94% (from 3.89% to 7.77%).

Table 5.3: Ablation Study on MR-Coupler (*DeepSeek-V3.1-Think*)

Metric	MR-Coupler	v_1: w/o func. context	v_2: w/o MTC expansion	v_3: w/o MTC validation
Num. of Generated TC	4151	3367	4324	4146
Pct. of Executable MTC	88.44%	63.86% (-27.80%)	88.34% (-0.10%)	91.65% (3.94%)
Pct. of Valid MTC	83.57%	**56.28% (-32.65%)**	81.38% (-2.62%)	83.04% (-0.63%)
Num. of Successful Task	98	**86 (-12.24%)**	98 (0.00%)	97 (-1.02%)
Pct. of False Alarm	5.50%	**11.86 (115.53%)**	5.73% (4.13%)	**9.39% (70.65%)**
Pct. of Bug-revealing MTC	7.77%	**3.37% (-56.62%)**	**3.89% (-49.92%)**	14.06% (81.04%)
Num. of Revealed Bugs	22	**12 (-45.45%)**	**13 (-40.91%)**	24 (9.09%)

Disabling *Validation and Filtering* (v_3) increased the false-alarm percentage by 70.65% (from 5.5% to 9.39%), indicating the effectiveness of the validation step in mitigating the false alarm issue. The slight increase in bug-revealing rate is because some bug-revealing MTCs are filtered out together with invalid ones. In some cases, both invalid and bug-revealing MTCs fail on both the original and mutated versions (0% pass rate), or valid MTCs fail to kill any mutants (100% pass rate on both), making mutation analysis based validation unable to distinguish them. A possible mitigation is to generate more diverse inputs to improve the mutant-killing capability.

Answer to RQ11: Each of the three steps uniquely enhances MR-Coupler's effectiveness. Functional coupling helps LLMs to generate more valid MTCs, MTC amplification augments the input to reveal more bugs, and mutation analysis based validation filters nearly half of the false alarms (reducing the rate from 9.39% to 5.5%).

5.2.6 RQ12: Similarity to human-written MTCs

Experiment Setup. Human-written MTCs represent well-established practices for constructing MRs, including the selection of method pairs as well as the input and output relation construction. This RQ evaluates whether the MTCs generated by MR-Coupler can mimic these practices by checking if they encode the same *MR-skeletons* as human-written ones. This can demonstrate the potential of MR-Coupler to assist developers in MTCs construction, facilitating developers in integrating the generated tests into their codebase and easing subsequent maintenance. This work uses the same 100 evaluation tasks as in RQ9.

According to the definition of MT in Section 2.1, an MR-skeleton consists of three core components: input relation, execution, and output relation. (i) *Input Relation:* the input transformation (e.g., API calls) applied to generate follow-up inputs, if applicable. (ii) *Execution:* the MR involved method pair (e.g., <`encryptText`, `decryptText`> in Listing 13). (iii) *Output Relation:* the assertion type (e.g., `assertEquals`) and the involved elements (e.g., source input, source output, follow-up output) to verify the output relation. For example, `assertEquals(plainText, decryptedText)` uses the assertion type `assertEquals`, with the involved elements source in-

Table 5.4: Similarity of MR-Coupler-generated MTCs to human-written MTCs

Metric	GPT-4o-mini		Qwen3-coder-Flash		Deepseek-V3.1		Deepseek-V3.1-Think		**Improv.**
	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	Baseline	MR-Coupler	
L1: Method-Pair Cons.	61%	89% (+45.90%)	61%	86% (+40.98%)	74%	87% (+17.56%)	81%	92% (+13.58%)	**29.51%**
L2: MR-Skeleton Cons.	46%	85% (+84.78%)	46%	84% (+82.61%)	55%	84% (+52.73%)	65%	90% (+38.46%)	**64.65%**

put (`plainText`) and follow-up output (`decryptedText`). Considering that the same output relation can be implemented in multiple ways, this work normalizes assertions for ease of comparison. Specifically, this work normalizes assertions to comparable assertions [3]. For example, boolean-style `assertTrue(x.equals(y))` and `assertFalse(x.equals(y))` are normalized to `assertEquals(x, y)` and `assertNotEquals(x, y)`, respectively. More details can be found in MR-Coupler's artifact [117].

Based on the definition of MR-skeleton, this work takes the human-written MTCs as the ground truth, and measure the similarity at two levels:

- L1: Method-Pair consistency: the proportion of target methods where at least one generated MTC couples the *same method pair* as the human-written MTC.
- L2: MR-Skeleton consistency: the proportion of target methods where at least one generated MTC encodes the same MR-skeleton as the human-written MTC, i.e., matching the input transformation, method pair, and output relation assertion type and elements. The MTCs satisfying L2 must satisfy L1 as well

Experiment Result. Table 5.4 shows that MR-Coupler-generated MTCs can match the human-coupled method pairs for 86∼92 target methods and encode the same MR-skeletons for 84∼90. Compared to the baseline, MR-Coupler improves method-pair consistency by 29.51% and full MR-skeleton consistency by 64.65%. These results highlight the effectiveness of MR-Coupler's coupling analysis based on patterns of relevant intention, similar implementation behavior, and potential state interference, MR-Coupler identified most functionally coupled methods used in human-written MTCs. The high MR-skeleton consistency further demonstrates MR-Coupler's potential to assist developers in MTCs construction, integration, and maintenance.

Despite the overall high consistency, MR-Coupler (*DeepSeek-V3.1-Think*) missed eight tasks in identifying the same method pairs and failed to encode the same MR-skeleton in ten tasks. Our inspection revealed three main causes: (i) some developer-selected method pairs exhibit implicit relevance in the code, such as `a2q` paired with `readAndWrite` for JSON serialization [133]; (ii) MR-Coupler-generated MTCs sometimes construct correct but different MRs, e.g., `cosineSimilarity` asserting equality for identical vectors versus inequality for distinct vectors [134]; and (iii) some inconsistencies arise from equivalent but differently expressed assertions, such as `assertEquals(x,y)` versus `assertTrue(a.equals(x)&&a.equals(y))` [135].

Table 5.5: Performance of MR-Coupler (GPT-4o-mini) on 100 Target Methods Before or After the Cut-Off Date

Target methods	Validity			Similarity	
	Num. of Successful Tasks	Pct. of Valid MTCs	Pct. of False alarm	L1: Method-Pair cons.	L2: MR-Skeleton cons.
Before Cut-off	92	71.84%	14.16%	89	85
After Cut-off	94	73.17%	9.52%	85	79

Answer to RQ12: The coupling analysis in MR-Coupler can identify most of the relevant methods used in human-written MTCs. MR-Coupler can achieve over 90% MR-skeleton consistency with human-written MTCs. This demonstrates MR-Coupler's potential to assist developers in MTC construction, facilitating developers in integrating and maintaining generated MTCs into their codebase.

5.2.7 Threats to Validity

Representativeness of LLMs. Since MR-Coupler relies on LLMs for MTC and input generation, one potential threat is whether our findings based on the selected LLMs are representative. To mitigate this threat, according to the EvalPlus leaderboard [93], this work includes representative LLMs from three well-known LLM families, i.e., *GPT-4o mini* from OpenAI [126],*Qwen3-coder-Flash* from Alibaba [92], *DeepSeek-V3.1* and *DeepSeek-V3.1-Think* from DeepSeek [91].

Data Contamination. A potential threat to our study is the data contamination issue, where some of the target programs or MTCs in our evaluation dataset may have been included in the pretraining data of the evaluated LLMs. If such memorization occurs, the models could gain an unfair advantage, thereby biasing the evaluation results. To mitigate this threat, this work followed the same collection procedure described in Section 5.2.1 to construct a *after-cutoff* dataset of entries after the training cutoff date of GPT-4o-mini (October 2023 [136]). As shown in Table 5.5, MR-Coupler achieved slightly lower similarity to human-written MTCs but a higher percentage of valid MTC compared to the pre-cutoff dataset. This indicates that MR-Coupler's effectiveness still holds for subjects after the cut-off date, and not simply an artifact of training-data memorization.

Representativeness of Subjects. A potential threat is whether our findings generalize to subjects of different projects. To mitigate this, this work adopted the strategy from existing studies [3, 49, 80] to include representative Java projects and evaluated MR-Coupler based on these projects (Section 5.2.1).

Quality of Ground Truths. This work uses human-written MTCs as ground truth, and take the fixed or latest versions of target programs as bug-free for answering RQs of validity, bug-revealing, and similarity. There is a potential threat regarding the quality of these ground truths. To mitigate this threat, this work applied three filtering criteria to select the valid and high-quality MTCs and corresponding target methods (Section 5.2.1).

5.3 Related Work

Constructing effective MRs is a critical step in conducting MT, and numerous approaches have been proposed to facilitate this process.

LLM-Based MR Generation. Recently, several studies have explored using LLMs to generate MRs. MR-Adopt [80] proposed an LLM-based test to automatically deduce the input relation from the pairs of hard-coded source and follow-up inputs. Shin *et al.* [102] employed LLMs to derive MRs from requirement specifications and translate them into an MR-specific language. Their approach works in two phases: (i) deriving metamorphic relations from a requirements document, and then (ii) converting MRs into SMRL (a domain-specific language for MRs). Zhang *et al.* [118] developed a human-AI hybrid MT framework that uses LLMs and predefined MR patterns to generate MRs for autonomous driving systems (ADSs). The two approaches are semi-automated and rely on human experts to select and refine generated MRs.

Some studies evaluated the effectiveness of LLMs in generating MR. Zhang *et al.* [123] conducted a pilot study using ChatGPT (3.5) for MR generation in ADS testing, and provided a methodology for generating MRs. Zhang *et al.* [124] evaluated LLMs (GPT-3.5 and GPT-4) on MR discovery across 37 subjects, finding that 4.6 38.6% of new MRs were rediscovered but only 29.9 43.8% of generated MRs were valid. These studies show the effectiveness of LLMs in generating MRs, but also challenges, such as a high invalidity rate of generated MRs.

Our approach (MR-Coupler) makes use of LLM's reasoning capability to come up with MRs based on functionally coupled methods and then generate concrete MTCs. MR-Coupler further mitigates the invalidity issue by validating generated MTCs via mutation analysis, providing an end-to-end automated and self-validating solution.

Traditional MR Generation Approaches. Prior to LLMs, most MR generation techniques relied on search-based, pattern-based, genetic-programming-based, or heuristic approaches. Ayerdi *et al.* [19, 103] and Terragni *et al.* [28] proposed approaches to generate MRs via genetic programming, but assuming the regression testing scenario. Zhang *et al.* [18] and Zhang *et al.* [17] proposed search-based approaches to inferring MRs for numeric programs. Zhou *et al.* [14] and Segura *et al.* [23]'s approaches identify MRs based on a set of predefined patterns. These approaches generated MRs are specific to domains or certain pre-defined patterns. Nolasco *et al.* [104] proposed MemoRIA to identify equivalence MRs from the documentation. Recently, Xu *et al.* [3] leveraged a new source (i.e., existing test cases) to automatically derive MRs. However, their approaches rely on rare resources (i.e., documents or tests embedded with MRs).

Compared with these approaches, our approach MR-Coupler is a fully automatic and domain-agnostic technique that generates MTCs directly from the target program. Not having the limitations of these approaches, MR-Coupler does not rely on resources like MR-embedded documents or tests, and is not restricted to the regression testing scenario.

5.4 Chapter Conclusion

This work presents MR-Coupler, a fully automated approach to generate MTCs directly from the target program via functional coupling analysis. MR-Coupler first identifies functionally coupled method pairs based on relevant intentions, similar implementation behavior, and potential state interference. It then employs LLMs to generate concrete MTCs and refines them based on execution feedback. Finally, MR-Coupler amplifies and then validates the MTCs based on mutation analysis.

The evaluation shows that MR-Coupler effectively generates valid MTCs for 98% of tasks and successfully reveals 22 confirmed bugs, improving valid MTC generation by 64.90%, and reducing false alarms by 36.56% compared to baselines. Moreover, MR-Coupler-generated achieves high consistency with human-written MR skeletons, demonstrating its potential to assist or even partially replace developers in constructing effective MTCs across diverse domains.

In summary, this work offers valuable insights into constructing metamorphic relations without relying on rare resources, such as human experts or high-quality specifications. It provides both a practical tool and a useful dataset for future research endeavors.

Data Availability. MR-Coupler and the experimental data are available at MR-Coupler's website [117].

CHAPTER 6

CONCLUSIONS

This thesis advances the automation of effective metamorphic testing (MT) by approaching one of its long-standing challenges: the construction of metamorphic relations (MRs). While prior research largely relied on manual derivation or domain-specific templates, this thesis proposed **three complementary approaches** that progressively reduce dependence on rare resource and leverage readily available software artifacts.

First, MR-Scout explored a novel direction of discovering and synthesizing MRs from existing developer-written test cases. MR-Scout demonstrated that valuable domain knowledge can be systematically extracted and reused. The resulting dataset of over 11,000 MR-encoded test cases across 701 projects provides the largest empirical foundation for relevant research.

Second, MR-Adopt addressed the problem of incomplete MRs by automatically deducing input transformation functions from ouput relations and examples. Framing the task as programming-by-example and leveraging large language models (LLMs) with program analysis, MR-Adopt enabled previously unusable MR-encoded tests to support automated new input generation, thereby enhancing test adequacy and reusability.

Third, MR-Coupler proposed a fully automatic method to generate metamorphic test cases via functional coupling analysis, requiring no existing MR-encoded tests or specifications. MR-Coupler successfully constructed high-quality MRs that uncovered real-world bugs, demonstrating the feasibility of deriving MRs directly from the source code.

These three approaches help transform MT from a knowledge-intensive technique into a largely automated and scalable process, supported by two released datasets of MR-encoded tests and MT-detectable real-world bugs.

REFERENCES

[1] Sergio Segura *et al.*, “A survey on metamorphic testing,” *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016 (cit. on pp. 1, 8, 20, 60, 65).

[2] Tsong Yueh Chen *et al.*, “Metamorphic testing: A review of challenges and opportunities,” *ACM Computing Surveys*, vol. 51, no. 1, 4:1–4:27, 2018 (cit. on pp. 1, 7, 8, 20, 34, 60).

[3] Congying Xu *et al.*, “Mr-scout: Automated synthesis of metamorphic relations from existing test cases,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 6, p. 150, 2024 (cit. on pp. 1, 40, 43, 51, 52, 58–60, 62, 63, 65, 66, 73, 76–78, 84, 85, 87).

[4] Vu Le *et al.*, “Compiler validation via equivalence modulo inputs,” in *Conference on Programming Language Design and Implementation*, ACM, 2014, pp. 216–226 (cit. on pp. 1, 40).

[5] Chengnian Sun *et al.*, “Finding compiler bugs via live code mutation,” in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications,*, ACM, 2016, pp. 849–863 (cit. on pp. 1, 40).

[6] Mikael Lindvall *et al.*, “Metamorphic model-based testing applied on NASA DAT - an experience report,” in *International Conference on Software Engineering*, IEEE Computer Society, 2015, pp. 129–138 (cit. on pp. 1, 40).

[7] Qiuyang Mang *et al.*, “Testing graph database systems via equivalent query rewriting,” in *International Conference on Software Engineering*, ACM, 2024, 143:1–143:12 (cit. on pp. 1, 40).

[8] Jialun Cao *et al.*, “Semmt: A semantic-based testing approach for machine translation systems,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, 34e:1–34e:36, 2022. doi: `10.1145/3490488` (cit. on pp. 1, 36, 40).

[9] Xiaoyuan Xie *et al.*, “Qaasker+: A novel testing method for question answering software via asking recursive questions,” *Automated Software Engineering*, vol. 30, no. 1, p. 14, 2023 (cit. on pp. 1, 40).

[10] Yuanyuan Yuan *et al.*, “Perception matters: Detecting perception failures of VQA models using metamorphic testing,” in *Conference on Computer Vision and Pattern Recognition*, Computer Vision Foundation / IEEE, 2021, pp. 16 908–16 917 (cit. on pp. 1, 40).

[11] John Ahlgren *et al.*, “Testing web enabled simulation at scale using metamorphic testing,” in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, IEEE, 2021, pp. 140–149. doi: `10.1109/ICSE-SEIP52600.2021.00023` (cit. on p. 1).

[12] Tsong Yueh Chen *et al.*, “METRIC: metamorphic relation identification based on the category-choice framework,” *J. Syst. Softw.*, vol. 116, pp. 177–190, 2016. doi: `10.1016/j.jss.2015.07.037` (cit. on pp. 1, 20, 36, 65).

[13] Chang-Ai Sun *et al.*, “Metric+: A metamorphic relation identification technique based on input plus output domains,” *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1764–1785, 2021. doi: `10.1109/TSE.2019.2934848` (cit. on pp. 1, 36).

[14] Zhi Quan Zhou *et al.*, “Metamorphic relations for enhancing system understanding and use,” *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1120–1154, 2020 (cit. on pp. 1, 20, 36, 63, 65, 87).

[15] Upulee Kanewala *et al.*, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, IEEE Computer Society, 2013, pp. 1–10. doi: `10.1109/ISSRE.2013.6698899` (cit. on pp. 1, 37).

[16] Arianna Blasi *et al.*, "Memo: Automatically identifying metamorphic relations in javadoc comments for test automation," *J. Syst. Softw.*, vol. 181, p. 111 041, 2021. doi: `10.1016/J.JSS.2021.111041` (cit. on pp. 1, 37, 65).

[17] Jie Zhang *et al.*, "Search-based inference of polynomial metamorphic relations," in *ACM/IEEE International Conference on Automated Software Engineering*, ACM, 2014, pp. 701–712 (cit. on pp. 1, 21, 27, 37, 63, 65, 87).

[18] Bo Zhang *et al.*, "Automatic discovery and cleansing of numerical metamorphic relations," in *IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2019, pp. 235–245 (cit. on pp. 1, 37, 63, 65, 77, 87).

[19] Jon Ayerdi *et al.*, "Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2021, pp. 1264–1274 (cit. on pp. 1, 37, 63, 65, 86).

[20] Jon Ayerdi *et al.*, "Evolutionary generation of metamorphic relations for cyber-physical systems," in *GECCO '22: Genetic and Evolutionary Computation Conference, Companion Volume, Boston, Massachusetts, USA, July 9 - 13, 2022*, Jonathan E. Fieldsend *et al.*, Eds., ACM, 2022, pp. 15–16. doi: `10.1145/3520304.3534077` (cit. on pp. 1, 65).

[21] Sergio Segura *et al.*, "A template-based approach to describing metamorphic relations," in *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, IEEE Computer Society, 2017, pp. 3–9. doi: `10.1109/MET.2017.3` (cit. on pp. 7, 8).

[22] T. Y. Chen *et al.*, "Metamorphic testing: A new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, Tech. Rep., 1998 (cit. on p. 7).

[23] Sergio Segura *et al.*, "Metamorphic testing of restful web apis," in *International Conference on Software Engineering*, ACM, 2018, p. 882 (cit. on pp. 11, 36, 60, 63, 65, 87).

[24] MR-Scout. "Mr-scout." (2023), [Online]. Available: `https://mr-scout.github.io` (cit. on pp. 13, 18, 22, 24, 25, 28, 31, 39).

[25] Kun Qiu *et al.*, "Theoretical and empirical analyses of the effectiveness of metamorphic relation composition," *IEEE Trans. Software Eng.*, vol. 48, no. 3, pp. 1001–1017, 2022. doi: `10.1109/TSE.2020.3009698` (cit. on pp. 14, 21, 36).

[26] Hengcheng Zhu *et al.*, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, IEEE, 2020, pp. 436–447. doi: `10.1145/3324884.3416539` (cit. on p. 15).

[27] Oracle. "Java language specification." (2023), [Online]. Available: `https://docs.oracle.com/javase/specs/` (cit. on p. 16).

[28] Valerio Terragni *et al.*, "Evolutionary improvement of assertion oracles," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu *et al.*, Eds., ACM, 2020, pp. 1178–1189. doi: `10.1145/3368089.3409758` (cit. on pp. 17, 22, 63, 66, 86).

[29] Junit. "Junit4." (2023), [Online]. Available: `https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html` (cit. on p. 18).

[30] Junit. “Junit5 assertions.” (2023), [Online]. Available: `https://junit.org/junit5/docs/5.0.3/api/org/junit/jupiter/api/Assertions.html` (cit. on p. 18).

[31] Carlos Pacheco *et al.*, “Randoop: Feedback-directed random testing for java,” in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2007, pp. 815–816 (cit. on pp. 19, 22, 44, 52).

[32] Gordon Fraser *et al.*, “Evosuite: Automatic test suite generation for object-oriented software,” in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2011, pp. 416–419 (cit. on pp. 19, 31, 38, 44, 52).

[33] Mark Harman *et al.*, “Achievements, open problems and challenges for search based software testing,” in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, IEEE Computer Society, 2015, pp. 1–12. doi: `10.1109/ICST.2015.7102580` (cit. on p. 22).

[34] EvoSuite. “Evosuite.” (2023), [Online]. Available: `https://www.evosuite.org/` (cit. on p. 22).

[35] Cristian Cadar *et al.*, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. doi: `10.1145/2408776.2408795` (cit. on p. 22).

[36] Gunel Jahangirova *et al.*, “Test oracle assessment and improvement,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller *et al.*, Eds., ACM, 2016, pp. 247–258. doi: `10.1145/2931037.2931062` (cit. on p. 22).

[37] Yun Lin *et al.*, “Graph-based seed object synthesis for search-based unit testing,” in *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis *et al.*, Eds., ACM, 2021, pp. 1068–1080. doi: `10.1145/3468264.3468619` (cit. on pp. 22, 27, 34).

[38] Alessio Gambi *et al.*, “SBST tool competition 2022,” in *15th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2022, Pittsburgh, PA, USA, May 9, 2022*, IEEE, 2022, pp. 25–32. doi: `10.1145/3526072.3527538` (cit. on pp. 22, 27, 34).

[39] GitHub. “Github.” (2023), [Online]. Available: `https://github.com/` (cit. on p. 23).

[40] Junit. “Junit5.” (2023), [Online]. Available: `https://junit.org/junit5/` (cit. on p. 23).

[41] TestNG. “Testng.” (2023), [Online]. Available: `https://testng.org/doc/` (cit. on p. 23).

[42] Hudson Borges *et al.*, “What’s in a github star? understanding repository starring practices in a social coding platform,” *J. Syst. Softw.*, vol. 146, pp. 112–129, 2018. doi: `10.1016/j.jss.2018.09.016` (cit. on p. 23).

[43] N Alan Heckert *et al.*, “Handbook 151: Nist/sematech e-handbook of statistical methods,” 2002 (cit. on pp. 26, 32).

[44] Pitest. “Pitest.” (2024), [Online]. Available: `https://pitest.org/` (cit. on pp. 29, 58).

[45] John A Rice, *Mathematical statistics and data analysis*. Cengage Learning, 2006 (cit. on pp. 29, 32).

[46] Andrea Arcuri *et al.*, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Softw. Test. Verification Reliab.*, vol. 24, no. 3, pp. 219–250, 2014. doi: `10.1002/STVR.1486` (cit. on pp. 29, 32).

[47] Gordon Fraser *et al.*, “Whole test suite generation,” 2, vol. 39, 2013, pp. 276–291. doi: `10.1109/TSE.2012.14` (cit. on p. 31).

[48] Pedro Delgado-Pérez *et al.*, “Interevo-tr: Interactive evolutionary test generation with readability assessment,” *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2580–2596, 2023. doi: `10.1109/TSE.2022.3227418` (cit. on p. 32).

[49] Ying Wang *et al.*, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *International Conference on Software Maintenance and Evolution*, IEEE, 2020, pp. 35–45 (cit. on pp. 34, 59, 85).

[50] Kaifeng Huang *et al.*, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022 (cit. on pp. 34, 59).

[51] Gordon Fraser *et al.*, "Evosuite: On the challenges of test case generation in the real world," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, IEEE Computer Society, 2013, pp. 362–369. doi: `10.1109/ICST.2013.51` (cit. on pp. 35, 36).

[52] Alastair F. Donaldson *et al.*, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, ACM, 2016, pp. 44–47. doi: `10.1145/2896971.2896978` (cit. on p. 36).

[53] Alastair F. Donaldson, "Metamorphic testing of android graphics drivers," in *Proceedings of the 4th International Workshop on Metamorphic Testing, MET@ICSE 2019, Montreal, QC, Canada, May 26, 2019*, Xiaoyuan Xie *et al.*, Eds., IEEE / ACM, 2019, p. 1. doi: `10.1109/MET.2019.00008` (cit. on p. 36).

[54] Dongwei Xiao *et al.*, "Metamorphic testing of deep learning compilers," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, 15:1–15:28, 2022. doi: `10.1145/3508035` (cit. on p. 36).

[55] Haoyang Ma *et al.*, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just *et al.*, Eds., ACM, 2023, pp. 248–260. doi: `10.1145/3597926.3598053` (cit. on pp. 36, 60).

[56] Matteo Paltenghi *et al.*, "Morphq: Metamorphic testing of the qiskit quantum computing platform," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, 2023, pp. 2413–2424. doi: `10.1109/ICSE48619.2023.00202` (cit. on p. 36).

[57] Leonhard Applis *et al.*, "Assessing robustness of ml-based program analysis tools using metamorphic program transformations," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, IEEE, 2021, pp. 1377–1381. doi: `10.1109/ASE51524.2021.9678706` (cit. on p. 36).

[58] Shuai Wang *et al.*, "Metamorphic object insertion for testing object detection systems," pp. 1053–1065, 2020. doi: `10.1145/3324884.3416584` (cit. on p. 36).

[59] Pingchuan Ma *et al.*, "Metamorphic testing and certified mitigation of fairness violations in NLP models," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, Christian Bessiere, Ed., ijcai.org, 2020, pp. 458–465. doi: `10.24963/ijcai.2020/64` (cit. on p. 36).

[60] Mikael Lindvall *et al.*, "Metamorphic model-based testing of autonomous systems," in *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, IEEE Computer Society, 2017, pp. 35–41. doi: `10.1109/MET.2017.6` (cit. on p. 36).

[61] Yongqiang Tian *et al.*, "To what extent do dnn-based image classification models make unreliable inferences?" *Empir. Softw. Eng.*, vol. 26, no. 4, p. 84, 2021. doi: `10.1007/s10664-021-09985-1` (cit. on p. 36).

[62] Chang-Ai Sun *et al.*, "Mumt: A data mutation directed metamorphic relation acquisition methodology," in *International Workshop on Metamorphic Testing*, ACM, 2016, pp. 12–18 (cit. on pp. 37, 65).

[63] Upulee Kanewala *et al.*, “Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels,” *Softw. Test. Verification Reliab.*, vol. 26, no. 3, pp. 245–269, 2016. doi: `10.1002/stvr.1594` (cit. on p. 37).

[64] Gordon Fraser *et al.*, “Generating parameterized unit tests,” in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer *et al.*, Eds., ACM, 2011, pp. 364–374. doi: `10.1145/2001420.2001464` (cit. on p. 38).

[65] Alexander Kampmann *et al.*, “Carving parameterized unit tests,” in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee *et al.*, Eds., IEEE / ACM, 2019, pp. 248–249. doi: `10.1109/ICSE-COMPANION.2019.00098` (cit. on p. 38).

[66] Suresh Thummalapenta *et al.*, “Retrofitting unit tests for parameterized unit testing,” in *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, Dimitra Giannakopoulou *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 6603, Springer, 2011, pp. 294–309. doi: `10.1007/978-3-642-19811-3\_21` (cit. on p. 38).

[67] Xiaoyuan Xie *et al.*, “Word closure-based metamorphic testing for machine translation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, 203:1–203:46, 2024. doi: `10.1145/3675396` (cit. on p. 40).

[68] Sumit Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *Symposium on Principles of Programming Languages, 2011*, ACM, 2011, pp. 317–330 (cit. on pp. 40, 57).

[69] Rangeet Pan *et al.*, “Can program synthesis be used to learn merge conflict resolutions? an empirical analysis,” in *IEEE/ACM International Conference on Software Engineering*, IEEE, 2021, pp. 785–796 (cit. on pp. 40, 57).

[70] Rajeev Alur *et al.*, “Syntax-guided synthesis,” in *Formal Methods in Computer-Aided Design*, IEEE, 2013, pp. 1–8 (cit. on pp. 40, 57).

[71] Daye Nam *et al.*, “Using an LLM to help with code understanding,” in *International Conference on Software Engineering*, ACM, 2024, 97:1–97:13 (cit. on p. 41).

[72] Mingyang Geng *et al.*, “Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning,” in *International Conference on Software Engineering*, ACM, 2024, 39:1–39:13 (cit. on p. 41).

[73] Lipeng Ma *et al.*, “Knowlog: Knowledge enhanced pre-trained language model for log understanding,” in *International Conference on Software Engineering*, ACM, 2024, 32:1–32:13 (cit. on p. 41).

[74] Junkai Chen *et al.*, “Code search is all you need? improving code suggestions with code search,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24, New York, NY, USA: Association for Computing Machinery, 2024 (cit. on p. 41).

[75] Maliheh Izadi *et al.*, “Language models for code completion: A practical evaluation,” in *International Conference on Software Engineering*, ACM, 2024, 79:1–79:13 (cit. on p. 41).

[76] Xueying Du *et al.*, “Evaluating large language models in class-level code generation,” in *nternational Conference on Software Engineering*, ACM, 2024, 81:1–81:13 (cit. on pp. 41, 46, 49, 54, 77).

[77] Michihiro Yasunaga *et al.*, “Large language models as analogical reasoners,” *CoRR*, vol. abs/2310.01714, 2023 (cit. on pp. 41, 64).

[78] Taylor Webb *et al.*, "Emergent analogical reasoning in large language models," *Nature Human Behaviour*, vol. 7, no. 9, pp. 1526–1541, 2023 (cit. on p. 41).

[79] Yujia Li *et al.*, "Competition-level code generation with alphacode," *CoRR*, vol. abs/2203.07814, 2022 (cit. on pp. 41, 45, 64, 65, 77).

[80] MR-Adopt. "Mr-adopt." (2024), [Online]. Available: `https://mr-adopt.github.io/` (cit. on pp. 43, 46, 48, 52, 53, 65, 71, 77, 85, 86).

[81] Jason Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Conference on Neural Information Processing Systems*, 2022 (cit. on p. 45).

[82] Zhiqiang Yuan *et al.*, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024 (cit. on pp. 45, 52, 60, 63–65, 71, 77).

[83] Zhen Yang *et al.*, "Exploring and unleashing the power of large language models in automated code translation," *CoRR*, vol. abs/2404.14646, 2024 (cit. on pp. 45, 52, 71).

[84] Ari Holtzman *et al.*, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, OpenReview.net, 2020 (cit. on pp. 46, 49, 54).

[85] Jialun Cao *et al.*, "Concerned with data contamination? assessing countermeasures in code language model," *CoRR*, vol. abs/2403.16898, 2024 (cit. on pp. 46, 49, 51, 54, 77).

[86] Mark Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021 (cit. on pp. 46, 49, 54).

[87] Aryaz Eghbali *et al.*, "De-hallucinator: Iterative grounding for llm-based code completion," *CoRR*, vol. abs/2401.01701, 2024 (cit. on pp. 46, 71).

[88] JavaParser. "Javaparser." (2024), [Online]. Available: `https://javaparser.org/` (cit. on pp. 49, 71).

[89] OpenAI. "Gpt-3.5." (2024), [Online]. Available: `https://platform.openai.com/docs/models/` (cit. on p. 52).

[90] Meta. "Llama-3-8b-instruct." (2024), [Online]. Available: `https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct` (cit. on p. 52).

[91] DeepSeek. "Deepseek-coder-7b-instruct-v1.5." (2024), [Online]. Available: `https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5` (cit. on pp. 52, 77, 85).

[92] Alibaba. "Codeqwen1.5." (2024), [Online]. Available: `https://qwenlm.github.io/blog/codeqwen1.5/` (cit. on pp. 52, 77, 85).

[93] Evalplus. "Leaderboard." (2024), [Online]. Available: `https://evalplus.github.io/leaderboard.html` (cit. on pp. 52, 77, 85).

[94] OpenAI. "Gpt-3.5 turbo updates." (2024), [Online]. Available: `https://help.openai.com/en/articles/8555514-gpt-3-5-turbo-updates` (cit. on p. 53).

[95] Meta. "Meta-llama-3 updates." (2024), [Online]. Available: `https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct` (cit. on p. 53).

[96] DeepSeek. "Deepseek-coder updates." (2024), [Online]. Available: `https://github.com/deepseek-ai/DeepSeek-Coder/issues/89` (cit. on p. 53).

[97] Chengshu Li *et al.*, "Chain of code: Reasoning with a language model-augmented code emulator," *CoRR*, vol. abs/2312.04474, 2023 (cit. on pp. 56, 64, 77).

[98] Yi Wu *et al.*, "How effective are neural networks for fixing security vulnerabilities," in *International Symposium on Software Testing and Analysis*, ACM, 2023, pp. 1282–1294 (cit. on p. 59).

[99] Songqiang Chen *et al.*, “Testing your question answering software via asking recursively,” in *International Conference on Automated Software Engineering*, IEEE, 2021, pp. 104–116 (cit. on p. 60).

[100] Huai Liu *et al.*, “How effectively does metamorphic testing alleviate the oracle problem?” *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014 (cit. on p. 60).

[101] Christos Tsigkanos *et al.*, “Variable discovery with large language models for metamorphic testing of scientific software,” in *Computational Science - ICCS 2023 - 23rd International Conference, Prague, Czech Republic, July 3-5, 2023, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 14073, Springer, 2023, pp. 321–335 (cit. on p. 63).

[102] Seung Yeob Shin *et al.*, “Towards generating executable metamorphic relations using large language models,” in *Quality of Information and Communications Technology - 17th International Conference on the Quality of Information and Communications Technology, QUATIC 2024, Pisa, Italy, September 11-13, 2024, Proceedings*, Antonia Bertolino *et al.*, Eds., ser. Communications in Computer and Information Science, vol. 2178, Springer, 2024, pp. 126–141. doi: `10.1007/978-3-031-70245-7\_9` (cit. on pp. 63, 65, 77, 86).

[103] Jon Ayerdi *et al.*, “Genmorph: Automatically generating metamorphic relations via genetic programming,” *IEEE Transactions on Software Engineering*, pp. 1–12, 2024 (cit. on pp. 63, 86).

[104] Agustín Nolasco *et al.*, “Abstraction-aware inference of metamorphic relations,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 450–472, 2024 (cit. on pp. 63, 77, 87).

[105] Chunqiu Steven Xia *et al.*, “Fuzz4all: Universal fuzzing with large language models,” in *International Conference on Software Engineering*, ACM, 2024, 126:1–126:13 (cit. on p. 63).

[106] Yutian Tang *et al.*, “Chatgpt vs sbst: A comparative assessment of unit test suite generation,” *IEEE Transactions on Software Engineering*, pp. 1–19, 2024 (cit. on pp. 63, 65, 77).

[107] Caroline Lemieux *et al.*, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *International Conference on Software Engineering*, IEEE, 2023, pp. 919–931 (cit. on p. 63).

[108] Chen Yang *et al.*, “Enhancing llm-based test generation for hard-to-cover branches via program analysis,” *CoRR*, vol. abs/2404.04966, 2024 (cit. on p. 63).

[109] Daya Guo *et al.*, “Deepseek-coder: When the large language model meets programming - the rise of code intelligence,” *CoRR*, vol. abs/2401.14196, 2024 (cit. on p. 64).

[110] Raymond Li *et al.*, “Starcoder: May the source be with you!” *CoRR*, vol. abs/2305.06161, 2023 (cit. on p. 64).

[111] Ziyang Luo *et al.*, “Wizardcoder: Empowering code large language models with evol-instruct,” *CoRR*, vol. abs/2306.08568, 2023 (cit. on p. 64).

[112] Bo Shen *et al.*, “Pangu-coder2: Boosting large language models for code with ranking feedback,” *CoRR*, vol. abs/2307.14936, 2023 (cit. on p. 64).

[113] Shihan Dou *et al.*, “Stepcoder: Improve code generation with reinforcement learning from compiler feedback,” *CoRR*, vol. abs/2402.01391, 2024 (cit. on p. 64).

[114] Ansong Ni *et al.*, “LEVER: learning to verify language-to-code generation with execution,” in *International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 26 106–26 128 (cit. on p. 64).

[115] Shuyang Jiang *et al.*, “Selfevolve: A code evolution framework via large language models,” *CoRR*, vol. abs/2306.02907, 2023 (cit. on p. 64).

[116] Kechi Zhang *et al.*, “Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges,” *CoRR*, vol. abs/2401.07339, 2024 (cit. on p. 64).

[117] MR-Coupler. “Mr-coupler.” (2025), [Online]. Available: https://mr-coupler.github.io/ (cit. on pp. 64, 67, 69, 84, 87).

[118] Yifan Zhang *et al.*, “Enhancing autonomous driving simulations: A hybrid metamorphic testing framework with metamorphic relations generated by GPT,” *Inf. Softw. Technol.*, vol. 187, p. 107 828, 2025. doi: 10.1016/J.INFSOF.2025.107828 (cit. on pp. 65, 77, 86).

[119] Ziyao Zhang *et al.*, “LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation,” *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 481–503, 2025. doi: 10.1145/3728894 (cit. on pp. 66, 68).

[120] TheAlgorithms. “Aesencryptiontest.” (2025), [Online]. Available: https://github.com/TheAlgorithms/Java/blob/master/src/test/java/com/thealgorithms/ciphers/AESEncryptionTest.java (cit. on p. 69).

[121] Apache Sedona. “Quadtreetest.” (2025), [Online]. Available: https://github.com/apache/sedona/blob/master/spark/common/src/test/java/org/apache/sedona/core/spatialPartitioning/quadtree/QuadTreeTest.java (cit. on p. 69).

[122] Eclipse EE4J. “Compactformattertest.java.” (2025), [Online]. Available: https://github.com/eclipse-ee4j/angus-mail/blob/master/providers/angus-mail/src/test/java/org/eclipse/angus/mail/util/logging/CompactFormatterTest.java (cit. on p. 69).

[123] Yifan Zhang *et al.*, “Automated metamorphic-relation generation with chatgpt: An experience report,” in *47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023*, IEEE, 2023, pp. 1780–1785. doi: 10.1109/COMPSAC57700.2023.00275 (cit. on pp. 70, 86).

[124] Jiaming Zhang *et al.*, “Can large language models discover metamorphic relations? A large-scale empirical study,” in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2025, Montreal, QC, Canada, March 4-7, 2025*, IEEE, 2025, pp. 24–35. doi: 10.1109/SANER64311.2025.00011 (cit. on pp. 70, 86).

[125] Major. “Mutation testing.” (2025), [Online]. Available: https://mutation-testing.org/ (cit. on pp. 73, 79).

[126] OpenAI. “Gpt-4o mini.” (2025), [Online]. Available: https://platform.openai.com/docs/models/gpt-4o-mini (cit. on pp. 77, 85).

[127] Gradoop Team. “Cnfestimation.” (2025), [Online]. Available: https://github.com/dbs-leipzig/gradoop/blob/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/matching/single/cypher/planning/estimation/CNFEstimation.java (cit. on p. 79).

[128] Jcabi. “Github commit 777a078913.” (2025), [Online]. Available: https://github.com/jcabi/jcabi-github/commit/777a078913 (cit. on p. 81).

[129] Brett Wooldridge. “Sparsebitset issue #13.” (2025), [Online]. Available: https://github.com/brettwooldridge/SparseBitSet/issues/13 (cit. on p. 81).

[130] Optimatika. “Ojalgo issue #49.” (2025), [Online]. Available: https://github.com/optimatika/ojAlgo/issues/49 (cit. on p. 81).

[131] Apache IoTDB. “Iotdb issue #13691.” (2025), [Online]. Available: https://github.com/apache/iotdb/pull/13691 (cit. on p. 81).

[132] Zingg. “Zingg issue #60.” (2025), [Online]. Available: https://github.com/zinggAI/zingg/issues/60 (cit. on p. 81).

[133] FasterXML. “Junit5testbase.” (2025), [Online]. Available: https://github.com/FasterXML/jackson-core/blob/2.x/src/test/java/com/fasterxml/jackson/core/JUnit5TestBase.java (cit. on p. 84).

[134] Diennea. “Simplerplannertest.” (2025), [Online]. Available: https://github.com/diennea/herddb/blob/master/herddb-core/src/test/java/herddb/sql/SimplerPlannerTest.java (cit. on p. 84).

[135] LocationTech. “Ntv2test.” (2025), [Online]. Available: https://github.com/locationtech/proj4j/blob/master/core/src/test/java/org/locationtech/proj4j/datum/NTV2Test.java (cit. on p. 84).

[136] [Online]. Available: https://community.openai.com/t/introducing-gpt-4o-mini-in-the-api/871594 (cit. on p. 85).