MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases

CONGYING XU, The Hong Kong University of Science and Technology, China
VALERIO TERRAGNI, The University of Auckland, New Zealand
HENGCHENG ZHU, The Hong Kong University of Science and Technology, China
JIARONG WU, The Hong Kong University of Science and Technology, China
SHING-CHI CHEUNG*, The Hong Kong University of Science and Technology, China

Metamorphic Testing (MT) alleviates the oracle problem by defining oracles based on metamorphic relations (MRs), that govern multiple related inputs and their outputs. However, designing MRs is challenging, as it requires domain-specific knowledge. This hinders the widespread adoption of MT. We observe that developer-written test cases can embed domain knowledge that encodes MRs. Such encoded MRs could be synthesized for testing not only their original programs but also other programs that share similar functionalities.

In this paper, we propose MR-Scout to automatically synthesize MRs from test cases in open-source software (OSS) projects. MR-Scout first discovers MR-encoded test cases (MTCs), and then synthesizes the encoded MRs into parameterized methods (called *codified MRs*), and filters out MRs that demonstrate poor quality for new test case generation. 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.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Software Testing, Metamorphic Testing, Metamorphic Relation, Automated Test Case Generation

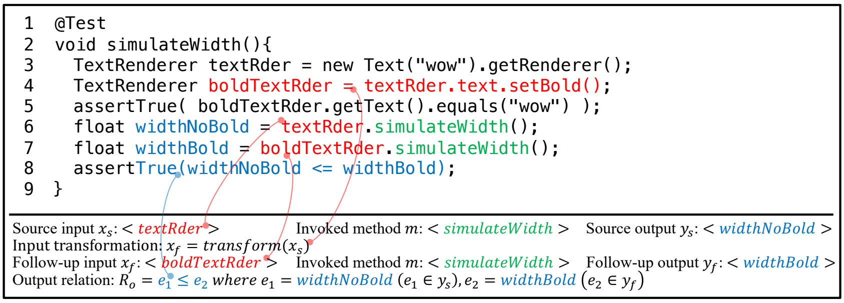
ACM Reference Format:
Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (March 2024), 28 pages. https://doi.org/---/---

1 INTRODUCTION

In recent years, automated test input generation has achieved significant advances [8, 17, 24, 40]. However, constructing test oracles is still a major obstacle to automated test case generation. **Metamorphic Testing (MT)** [10] has been applied to various domains as a promising approach to addressing the test oracle problem [46]. MT works by employing additional test inputs when the expected output for a given input is difficult to determine. It reveals a fault if a relation (known as a **Metamorphic Relation (MR)**) between these inputs and their corresponding outputs is violated. For instance, consider a program $P(a, b, G)$ that computes the shortest path from vertex a to vertex b in a large undirected graph G. It is difficult to determine the expected output of $P(a, b, G)$, while a correct implementation of P should provide the same length of paths for $P(a, b, G)$ and $P(b, a, G)$. Validating whether the outputs of $|P(a, b, G)|$ and $|P(b, a, G)|$ are equal is much easier

[*] Shing-Chi Cheung is the corresponding author.

Authors' addresses: Congying Xu, cxubl@connect.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Valerio Terragni, v.terragni@auckland.ac.nz, The University of Auckland, Auckland, New Zealand; Hengcheng Zhu, hzhuaq@connect.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Jiarong Wu, jwubf@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Shing-Chi Cheung, scc@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China.



```
@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: $R_o = e_1 \leq e_2$ *where* e_1 = *widthNoBold* ($e_1 \in y_s$), e_2 = *widthBold* ($e_2 \in y_f$)

Fig. 1. A test case crafted from `com.itextpdf.layout.renderer.TextRendererTest` in project iText. Underlying MR: *IF* $text_2 = text_1.setBold()$ *THEN* $text_1.width() \leq text_2.width()$.

(the associated MR is $|P(a, b, G)| = |P(b, a, G)|$). An advantage of MT is that an MR can serve as an oracle that is applicable to many test inputs. It enables automated test case generation by integrating MRs with automatically generated test inputs [46]. However, the design of MRs is challenging because it requires domain-specific knowledge and relies on the expertise of testers [1]. This hinders the wider adoption of MT [46].

There exist studies trying to systematically explore MRs, such as identifying MRs from software specifications [12, 48] and searching MRs using pre-defined patterns [47, 60] 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 [6, 32], search-based approaches [58, 59], and genetic-programming-based approaches [4, 5]. However, wide adoption of these approaches is challenging. They are designed for programs in specific domains (e.g., numerical programs [58, 59]) whose input and output values exhibit certain types of relations (e.g., equivalence relations [6], polynomial relations [58], or relations that follow pre-defined patterns [4]).

Our Observations and Idea. We observe 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. We refer to such test cases as *MR-encoded test cases* (MTCs). For example, the test case `simulateWidth()` in Figure 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 [47]. This observation motivates us to design an automatic approach to synthesize MRs from existing test cases for automated test case generation.

Challenge. 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 1, there is no documentation of the encoded MR in either comments or annotations. After understanding the logic of this test case, we can recognize the underlying MR and its corresponding constituents. 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.

Methodology. In this paper, 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. Since an MR is defined over at least two inputs and corresponding outputs, we derive 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, *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 3.2, *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 3.3, *MR Filtering*).

Evaluation. We 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, we 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 our MTC dataset (Section 4.2, RQ1). MR-Scout synthesizes codified MRs from MTCs and applies filtering to remove low-quality MRs. To evaluate the effectiveness of this process, we 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 4.3, RQ2). Furthermore, to demonstrate the usefulness of synthesized MRs in complementing existing tests and enhancing test adequacy, we 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 4.4, RQ3). To evaluate the comprehensibility of codified MRs, we 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.

Contribution. Our work makes the following contributions.

- We propose MR-Scout, the first approach that automatically synthesizes MRs from existing test cases.
- We release a dataset of over 11,000 MTCs discovered across 701 OSS projects, and investigate 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.
- We conduct 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.
- We release the research artifact and all experimental datasets on MR-Scout's website [54] to facilitate reproducing our experimental results and future research.

2 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 $\langle f(x_1), \cdots, f(x_n)\rangle$ [11]. Following the definition by Segura et al. [45], 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. [10]. It expresses an MR in terms of an input relation and an output relation.

Example 2.1. 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, $\mathcal{R}_i = \{((v_1, v_2, G), (v_2, v_1, G)) \mid \forall G, \forall v_1, v_2 \in G\}$ includes all pairs of inputs to f such that the first two elements (source and sink vertexes) are swapped. $\mathcal{R}_o = \{(n, n) \mid \forall n \in \mathbb{N}\}$ includes all pairs of equivalent numbers (shortest paths).

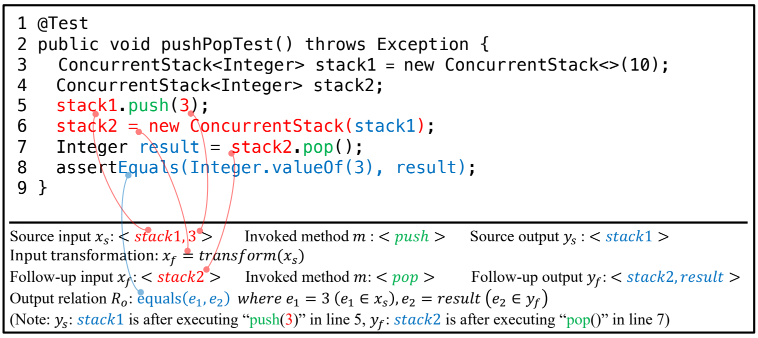
Metamorphic Testing (MT). Given an MR $\mathcal{R}$ for a function f, metamorphic testing is the process of validating $\mathcal{R}$ on an implementation P of f using various inputs [45].

Intuitively, assuming a program implemented by a sequence of statements, MT entails the following five steps [11]: (i) constructing a source input, which can be written by developers or automatically generated (e.g., random testing) [46], (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 $\mathcal{R}_o$.

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 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 1), our goal is to automatically discover and synthesize these encoded MRs from existing test cases in open-source projects. This paper focuses on unit test cases for object-oriented programming (OOP) programs. Since the existing MR formulation is not originally designed for OOP programs,



```
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: $\mathtt{equals}(e_1, e_2)$ $where\ e_1 = 3\ (e_1 \in x_s), e_2 = result\ (e_2 \in y_f)$
(Note: y_s: $stack1$ is after executing "push(3)" in line 5, y_f: $stack2$ is after executing "pop()" in line 7)

Fig. 2. 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.

```
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()
}
```

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.)

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, 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.

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

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

Example 2.2. Let f_c stand for the class under test `ConcurrentStack` in Figure 2. Given the illustration in Listing 1, the relation $x = stack.push(x).pop()$ can be expressed as: *IF* two inputs

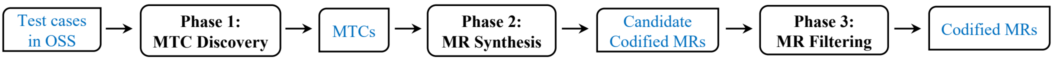


Fig. 3. Overview of MR-Scout

$\langle x_1, x_2 \rangle$ 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 2.3. When the function f_c wraps the class `TextRenderer` in Figure 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 $\mathcal{R}_o$.

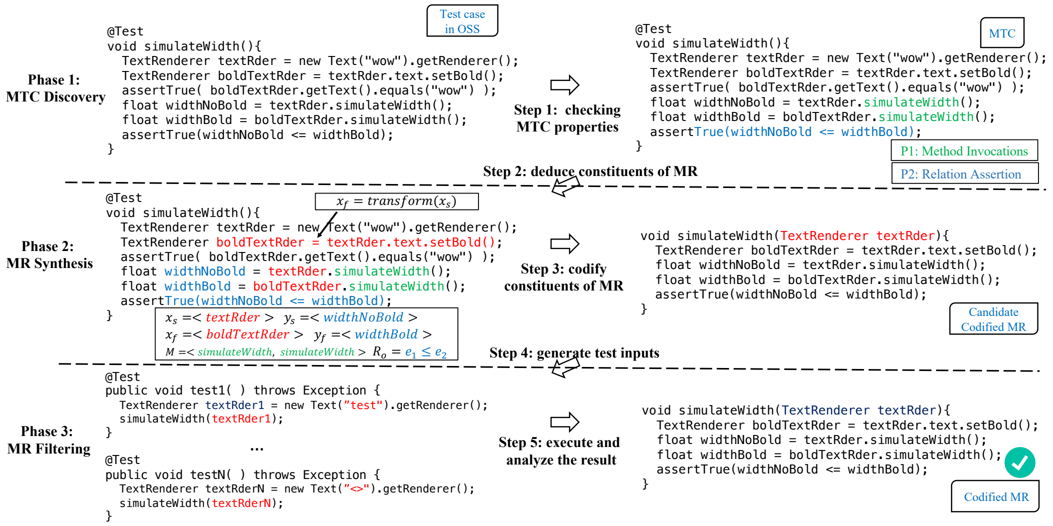
Note that, for methods with nested method calls, such as $m(g(\cdot))$, the nested method call $g(\cdot)$ is considered as an argument to the method m. $m(g(\cdot))$ can be expressed as $m(x)$ where $x.arg = g(\cdot)$. For methods having side effects, the identification of such methods' outputs will be discussed in Section 3.1.2.

3 METHODOLOGY

Inspired by our observation that test cases written by developers can embed domain knowledge that encodes MRs, we propose 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 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, we derive two principal 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).

(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 paper (Section 3.2).



```
@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);
}
```

```
@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);
}
```

Fig. 4. Procedure of MR-Scout operating on the MTC `simulateWidth()`

Table 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`));`

(3) *MR Filtering*. We target discovering MRs for new test generation. Codified MRs not applicable to new test inputs are ineffective for new test generation [43]. 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 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. Therefore, to discover possible MTCs, MR-Scout should analyze whether the given test cases embed the semantics of an MR. So we first model 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.

3.1.1 *Properties of An MTC.*

According to the formulation of MR in Section 2.1, we derive 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, we allow the invocations to the same or different methods of a class under test.

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 $\mathcal{R}_o$.

3.1.2 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 [61]. 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, we propose 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.* We first introduce the necessary elements of an output relation, and then introduce 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 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, 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, we discuss what are the input x_i and output y_i of a method invocation mi_i. According to the specification of Java [39], 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. For the test case in Figure 2, there are two method invocation mi_1 = `stack1.push(3)` on line 5 and mi_2 =`stack2.pop()` on line 7, where x_1 = {`stack1`, 3}, x_2 = {`stack2`}, y_1 = {`stack1`} (just after `stack1.push(3)`), and y_2 = {`result`, `stack2`} (just after `stack2.pop()`). The assertion α on line 8 can be interpreted as `assertEquals (Integer.valueOf(`e_1`),`e_2`)`, where $e_1 = 3$ ($e_1 \in x_1$) and e_2 = `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, we further check 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 [50], 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., =, <, >, ≤, ≥, ≠), or (ii) user-defined methods that return boolean values.

Example 3.2. The assertion `assertTrue(widthNoBold <= widthBold)` in Figure 1 can be mapped onto *A1-BoolAssert* , where e_1 = `widthNoBold` and e_2 = `widthBold` are related by a numerical operator "≤".

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 3.3. The assertion `assertEquals(Integer.valueOf(3), result)` in Figure 2 can be mapped onto *A2-CompAssert*, where e_1 = 3 and e_2 = `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 1, the corresponding APIs in JUnit4 [28] 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. We consider 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 $\langle \alpha, MI \rangle$, 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.

3.1.3 *Detailed Analysis Process and Limitations.*

MR-Scout [54] 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.

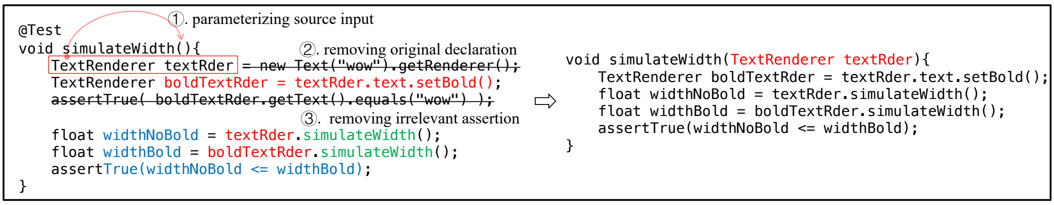
3.2 Phase 2: MR Synthesis

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 our knowledge, no existing study explores this problem, nor does our paper. In this paper, we focus 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 [40] and EvoSuite [17]) 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.

3.2.1 *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 3.1.2), for each MR instance $\langle \alpha, MI \rangle$, 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. parameterizing source input
②. removing original declaration
③. removing irrelevant assertion
⇨

```
@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);
}
```

Fig. 5. Illustration of constructing a codified MR

(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 $x_2 = transform(s)$ ($s \subseteq x_1 \cup y_1$), 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 paper 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 [11, 12, 46, 60]. Our evaluation results reveal that 64.13% of MR instances only involve two method invocations (Section 4.1.1), 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 3.4. In Phase 2 of Figure 4, 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)`.

3.2.2 *Step 3: Codify Constituents of MR.*

This step mainly consists of parameterizing the source input and removing irrelevant assertions. We illustrate the process of constructing a codified MR using the example shown in Figure 5.

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 5, 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 5).

Next, MR-Scout removes irrelevant assertions (③ in Figure 5). Assertions not identified as relation assertions are considered irrelevant and removed. These irrelevant assertions may be specific to the original value of the source input and could lead to false alarms when new inputs are introduced. In Figure 5, 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.

3.3 Phase 3: MR Filtering

We aim 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 [43]. 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 [59], 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, we are 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, we observe 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 our 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 [40], search based [16, 24], and symbolic execution based techniques [8]. Following existing works on test oracle assessment and improvement [27, 50], MR-Scout employs EvoSuite [16] 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 we give the same time budget rather than the same amount of inputs for each codified MR. In line with the configuration of previous works [21, 34], 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 [54].

Then, MR-Scout executes these test cases (as illustrated in Figure 4 (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.

4 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 4.2)
RQ2 Quality: How is the quality of MR-Scout codified MRs in applying to new inputs for automated test case generation? (Section 4.3)
RQ3 Usefulness: How useful are MR-Scout codified MRs in enhancing test adequacy? (Section 4.4)
RQ4 Comprehensibility: Are MRs codified by MR-Scout comprehensible? (Section 4.5)

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, we first ran MR-Scout, and 11,350 MTCs from 701 projects were discovered. Then, we 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, we analyze 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, we conducted a small-scale qualitative study on 52 codified MRs.

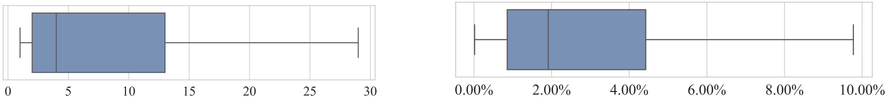
4.1 Dataset Preparation

We selected open-source projects from GitHub [22]. We 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 [29] and TestNG [51]. The number of stars indicates the popularity and correlates with project quality [7]. We 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. We cloned the latest version of each selected project at that time and collected tests from these projects. We considered methods annotated with "`@Test`" as tests and files containing tests as test files. We excluded 3,327 projects without tests. At last, we 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.

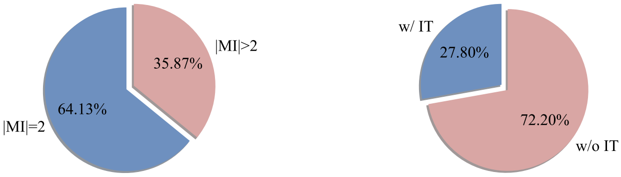
4.1.1 MTC Discovery. We ran MR-Scout on each of the 4,068 projects on a machine with dual Intel® Xeon™ E5-2683 v4 CPUs and 256 GB system memory. For the *MTC Discovery* phase, MR-Scout took 18 hours and 58 minutes, with an average analysis time of 16.78 seconds per project. Finally, 11,350 MTCs in 701 (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.



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

Fig. 6. Distribution of 11,350 MR-encoded test cases (MTCs) in 701 projects w.r.t the number and percentage



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

Fig. 7. Distribution of 21,574 MR instances w.r.t the size of involved method invocations (|MI|) and the existence of an input transformation (IT)

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 6a, 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. We further analyzed the percentage of MTCs among all tests in each project in Figure 6. 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%.

We also examined the top 25 projects with the highest number of MTCs (the projects can be found at [54]). 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 3.2). 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 transformation. In this work, we target 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 [54].

4.1.2 *MR Synthesis and Filtering.*

In this paper, we focus 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, we 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 5.

4.2 RQ1: Precision

4.2.1 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, we aim to evaluate whether MR-Scout is precise in discovering MR-encoded test cases in real-world projects. The results also reflect the reliability of our released dataset of discovered MTCs.

To achieve this goal, we manually validate if the MTCs discovered by MR-Scout have the two properties mentioned in Section 3.1. However, there are 11,350 MTCs discovered in our evaluation subjects, and it is infeasible to check all of them manually. Therefore, we randomly sample 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 [25].

During the validation, two authors of this paper 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.
- *False Positive* indicates a test case does not possess the two properties.
- *Unclear* indicates the author cannot understand a test case or tell if the two properties are possessed.

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

4.2.2 Result. During the manual validation, there were 27 test cases assigned with different labels by the two authors, 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)`

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

```
...
    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
...
```

Listing 2. Simplified example of a false positive MTC

which are not methods in the class under test. `AssertEquals(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. 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.

4.3 RQ2: Quality

4.3.1 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 [59], 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, we aim 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, we reuse the criterion of **a high-quality MR** defined in Section 3.3. 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, we evaluate the quality of 75 codified MRs that are output by MR-Scout after filtering (Section 4.1.2). In line with the configuration of the previous studies [21, 34], we re-ran EvoSuite 10 times with different seeds to mitigate the randomness issue on the evaluation results and gave a time budget of 2 minutes for each run[2]. With the generated inputs, we 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 3.3. 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 8 shows the distribution of generated test inputs.

4.3.2 Result. Out of 71 MR-Scout output MRs, we 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,

[2] The detailed configuration of EvoSuite can be found on MR-Scout's website [54].

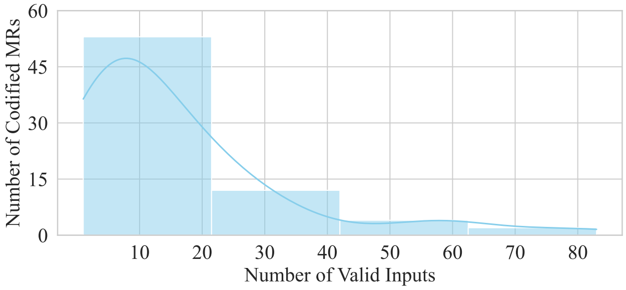


Fig. 8. Distribution of generated valid inputs

we 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.

4.4 RQ3: Usefulness

4.4.1 Experiment Setup. In this RQ, we aim 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 we focus on assessing how useful codified-MRs-based tests are in complementing existing tests and improving the test adequacy of these original programs.

Metrics and Baselines. We employ 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.

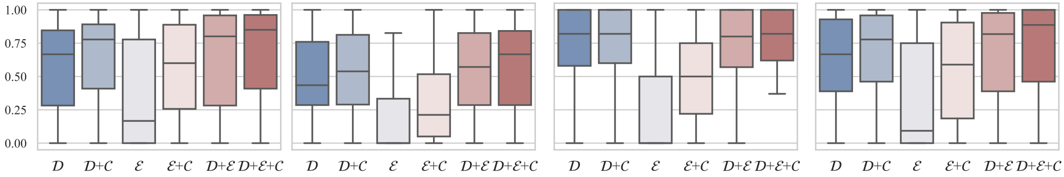
Firstly, codified MRs are integrated with automatically generated valid inputs in the RQ2 (Quality) to construct codified-MR-based test suites (denoted as $\mathcal{C}$). Then, we compare the performance of the codified-MR-based test suite on these four metrics against two baselines: (i) developer-written 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, we do not directly compare the performance of the codified-MR-based test suite against developer-written or EvoSuite-generated test suites. Instead, we investigate whether the codified-MR-based test suites can enhance the test adequacy on top of developer-written and EvoSuite-generated test suites.

We successfully ran PIT [42], 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 4.1.2). There are a total of 4,701 lines of code in these target classes.

Statistical Analysis. We performed a statistical analysis (i.e., Mann-Whitney U-test [3, 44]) 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, we compare the fault

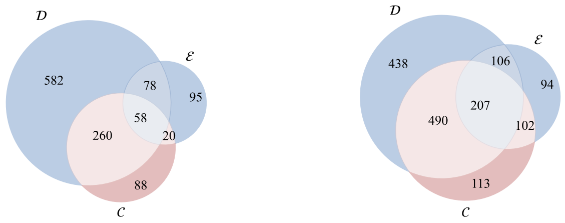
Table 2. Enhancement of test adequacy by codified-MR-based test suites ($\mathcal{C}$) on top of developer-written ($\mathcal{D}$) and EvoSuite-generated test suites ($\mathcal{E}$)

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**	$\mathcal{D}$+$\mathcal{E}$	$\mathcal{D}$+$\mathcal{E}$+$\mathcal{C}$	**Enhancement**
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%



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

Fig. 9. 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

Fig. 10. Comparison of covered and killed mutants by developer-written ($\mathcal{D}$), EvoSuite-generated ($\mathcal{E}$), and codified-MR-based ($\mathcal{C}$) test suites

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.

4.4.2 Result. Table 2 presents the results of the four metrics on 51 classes. Compared with $\mathcal{D}$, 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 9 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 10 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 [17, 19].

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 [17]. 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 [54]. 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.

4.5 RQ4: Comprehensibility

4.5.1 Experiment Setup. We consider 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, we design RQ4 to assess the comprehensibility of codified MRs synthesized by MR-Scout.

To this end, we 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, we randomly sampled 52 cases from the 75 MR-Scout synthesized codified MRs (which were collected in the dataset preparation, Section 4.1.2) 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 [25].

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 [13] (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 [3, 44]) 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.

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

4.5.2 Result. Figure 11 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.

We also gathered feedback from participants to investigate the factors that make synthesized MRs difficult to understand. We found that the main difficulty in understanding some MRs is from the complexity of certain classes under test. The test cases in our evaluation were collected from highly-starred Java projects, which often exhibit complex structural dependencies between classes (Section 4.1.1). In our 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.

5 DISCUSSION

5.1 Threats to Validity

We have identified potential threats to the validity of our experiments and have 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, we gave the participants a training session before manual validation. for RQ1, two authors independently

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

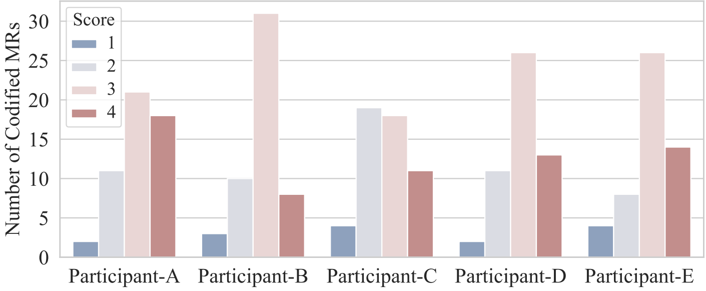


Fig. 11. Comprehensibiliy scores of 52 MR-Scout synthesized MRs (Score: 1. very difficult, 2. difficult, 3. easy 4. every easy to understand)

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, we first adopted criteria from previous empirical studies on OSS projects [26, 56] to select high-quality and well-maintained Java projects, as described in Section 4.1.1. Then, we quantified the coverage of our selected projects to all popular Java projects on GitHub. The result shows that our 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 our 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, we followed the practices of existing studies [21, 34] to run EvoSuite 10 times and chose appropriate parameters that fit our scenario.

5.2 Applications of MR-Scout

Metamorphic testing is an approach to both test result verification (i.e., test oracle problem) and test case generation [11] 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 4.4: Usefulness). Furthermore, codified MRs, representing properties of target programs, help describe the behaviors of classes under test across potential test inputs, simplifying test maintenance.

5.3 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 4.2. 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, we 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 [18]: (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 [18], "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.

6 RELATED WORK

6.1 MR Identification

Many studies proposed MRs for testing programs of various domains (e.g., compilers [14, 15, 36, 57], quantum computing [41], and AI systems [2, 9, 35, 37, 53, 55]). We review and discuss the most closely related work in systematically identifying MR.

MR Pattern Based Approaches. Segura et al. [47] proposed six MR output patterns for Web APIs, and a methodology for users to identify MRs. Similarly, Zhou et al. [60] 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+ [48], 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. [43] 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. [59] 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. [58] 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. [4] proposed GAssertMRs, 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 GAssertMRs are limited to three pre-defined MR Input Patterns. Sun et al. [49] 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 [32] 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. [33] 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. [6] 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.

6.2 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 [17]) 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. [20] 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. [31] assumed the existence of high-quality system tests and proposed to automatically extract parameterized unit tests from system test executions. Thummalapenta et al. [52] 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. [52] 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.

7 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, we introduce MR-Scout to automatically discover and synthesize MRs from existing test cases in OSS projects. We model 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.

8 DATA AVAILABILITY AND STATEMENT

Data Availability. We make MR-Scout and the experimental data publicly available at MR-Scout's site [54] to facilitate the reproduction of our study and relevant studies of other researchers in the community.

Statement of AI Tool Usage. During the paper writing, we used Grammarly [23] and ChatGPT [38] to check grammar.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation of China (Grant No. 61932021), the Hong Kong Research Grant Council/General Research Fund (Grant No. 16207120), and the Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034-18).

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. 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, 140–149. https://doi.org/10.1109/ICSE-SEIP52600.2021.00023

[2] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. 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, 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706

[3] Andrea Arcuri and Lionel C. Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verification Reliab.* 24, 3 (2014), 219–250. https://doi.org/10.1002/STVR.1486

[4] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. 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, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1264–1274. https://doi.org/10.1145/3468264.3473920

[5] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2022. 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 and Markus Wagner (Eds.). ACM, 15–16. https://doi.org/10.1145/3520304.3534077

[6] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/j.jss.2021.111041

[7] Hudson Borges and Marco Túlio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[9] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. 2022. SemMT: A Semantic-Based Testing Approach for Machine Translation Systems. *ACM Trans. Softw. Eng. Methodol.* 31, 2 (2022), 34e:1–34e:36. https://doi.org/10.1145/3490488

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology.

[11] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27. https://doi.org/10.1145/3143561

[12] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *J. Syst. Softw.* 116 (2016), 177–190. https://doi.org/10.1016/j.jss.2015.07.037

[13] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. 2023. InterEvo-TR: Interactive Evolutionary Test Generation With Readability Assessment. *IEEE Trans. Software Eng.* 49, 4 (2023), 2580–2596. https://doi.org/10.1109/TSE.2022.3227418

[14] Alastair F. Donaldson. 2019. 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, Pak-Lok Poon, and Laura L. Pullum (Eds.). IEEE / ACM, 1. https://doi.org/10.1109/MET.2019.00008

[15] Alastair F. Donaldson and Andrei Lascu. 2016. 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, 44–47. https://doi.org/10.1145/2896971.2896978

[16] EvoSuite. 2023. *EvoSuite*. Retrieved August 20, 2023 from https://www.evosuite.org/

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[18] Gordon Fraser and Andrea Arcuri. 2013. 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, 362–369. https://doi.org/10.1109/ICST.2013.51

[19] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2, 276–291. https://doi.org/10.1109/TSE.2012.14

[20] Gordon Fraser and Andreas Zeller. 2011. 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 and Frank Tip (Eds.). ACM, 364–374. https://doi.org/10.1145/2001420.2001464

[21] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. 2022. 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, 25–32. https://doi.org/10.1145/3526072.3527538

[22] GitHub. 2023. *GitHub*. Retrieved August 20, 2023 from https://github.com/

[23] Grammarly. 2023. *Grammarly*. Retrieved August 20, 2023 from http://grammarly.com

[24] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. 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, 1–12. https://doi.org/10.1109/ICST.2015.7102580

[25] N Alan Heckert, James J Filliben, C M Croarkin, B Hembree, William F Guthrie, P Tobias, and J Prinz. 2002. Handbook 151: Nist/sematech e-handbook of statistical methods. (2002).

[26] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empir. Softw. Eng.* 27, 4 (2022), 90. https://doi.org/10.1007/s10664-022-10131-8

[27] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. 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 and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[28] Junit. 2023. *Junit4*. Retrieved August 20, 2023 from https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html

[29] Junit. 2023. *Junit5*. Retrieved August 20, 2023 from https://junit.org/junit5/

[30] Junit. 2023. *Junit5 Assertions*. Retrieved August 20, 2023 from https://junit.org/junit5/docs/5.0.3/api/org/junit/jupiter/api/Assertions.html

[31] Alexander Kampmann and Andreas Zeller. 2019. 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, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 248–249. https://doi.org/10.1109/ICSE-COMPANION.2019.00098

[32] Upulee Kanewala and James M. Bieman. 2013. 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, 1–10. https://doi.org/10.1109/ISSRE.2013.6698899

[33] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. 2016. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Softw. Test. Verification Reliab.* 26, 3 (2016), 245–269. https://doi.org/10.1002/stvr.1594

[34] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. 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, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1068–1080. https://doi.org/10.1145/3468264.3468619

[35] Mikael Lindvall, Adam A. Porter, Gudjon Magnusson, and Christoph Schulze. 2017. 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, 35–41. https://doi.org/10.1109/MET.2017.6

[36] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. , 248–260 pages. https://doi.org/10.1145/3597926.3598053

[37] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. 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, 458–465. https://doi.org/10.24963/ijcai.2020/64

[38] OpenAI. 2023. *ChatGPT.* Retrieved August 20, 2023 from https://openai.com/blog/chatgpt

[39] Oracle. 2023. *Java Language Specification.* Retrieved August 20, 2023 from https://docs.oracle.com/javase/specs/

[40] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 815–816. https://doi.org/10.1145/1297846.1297902

[41] Matteo Paltenghi and Michael Pradel. 2023. 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, 2413–2424. https://doi.org/10.1109/ICSE48619.2023.00202

[42] PITest. 2023. *PITest.* Retrieved August 20, 2023 from https://pitest.org/

[43] Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. 2022. Theoretical and Empirical Analyses of the Effectiveness of Metamorphic Relation Composition. *IEEE Trans. Software Eng.* 48, 3 (2022), 1001–1017. https://doi.org/10.1109/TSE.2020.3009698

[44] John A Rice. 2006. *Mathematical statistics and data analysis.* Cengage Learning.

[45] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. 2017. 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, 3–9. https://doi.org/10.1109/MET.2017.3

[46] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[47] Sergio Segura, José Antonio Parejo, Javier Troya, and Antonio Ruiz Cortés. 2018. Metamorphic testing of RESTful web APIs. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 882. https://doi.org/10.1145/3180155.3182528

[48] Chang-Ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. 2021. METRIC^{+}+: A Metamorphic Relation Identification Technique Based on Input Plus Output Domains. *IEEE Trans. Software Eng.* 47, 9 (2021), 1764–1785. https://doi.org/10.1109/TSE.2019.2934848

[49] Chang-Ai Sun, Yiqiang Liu, Zuoyi Wang, and W. K. Chan. 2016. μMT: a data mutation directed metamorphic relation acquisition methodology. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016.* ACM, 12–18. https://doi.org/10.1145/2896971.2896974

[50] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. 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, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1178–1189. https://doi.org/10.1145/3368089.3409758

[51] TestNG. 2023. *TestNG.* Retrieved August 20, 2023 from https://testng.org/doc/

[52] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. 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 (Lecture Notes in Computer Science, Vol. 6603)*, Dimitra Giannakopoulou and Fernando Orejas (Eds.). Springer, 294–309. https://doi.org/10.1007/978-3-642-19811-3_21

[53] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang. 2021. To what extent do DNN-based image classification models make unreliable inferences? *Empir. Softw. Eng.* 26, 4 (2021), 84. https://doi.org/10.1007/s10664-021-09985-1

[54] MR-Scout. 2023. *MR-Scout.* Retrieved August 20, 2023 from https://mr-scout.github.io

[55] Shuai Wang and Zhendong Su. 2020. Metamorphic Object Insertion for Testing Object Detection Systems. (2020), 1053–1065. https://doi.org/10.1145/3324884.3416584

[56] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020.* IEEE, 35–45. https://doi.org/10.1109/ICSME46990.2020.00014

[57] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1 (2022), 15:1–15:28. https://doi.org/10.1145/3508035

[58] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019.* IEEE, 235–245. https://doi.org/10.1109/ICSME.2019.00035

[59] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based inference of polynomial metamorphic relations. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14,*

Vasteras, Sweden - September 15 - 19, 2014, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 701–712. https://doi.org/10.1145/2642937.2642994

[60] Zhi Quan Zhou, Liqun Sun, Tsong Yueh Chen, and Dave Towey. 2020. Metamorphic Relations for Enhancing System Understanding and Use. *IEEE Trans. Software Eng.* 46, 10 (2020), 1120–1154. https://doi.org/10.1109/TSE.2018.2876433

[61] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. 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, 436–447. https://doi.org/10.1145/3324884.3416539