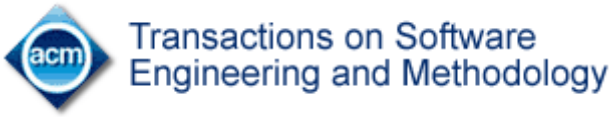


MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases

Journal:	*Transactions on Software Engineering and Methodology*
Manuscript ID	TOSEM-2023-0286
Manuscript Type:	Journal-First Paper
Date Submitted by the Author:	23-Aug-2023
Complete List of Authors:	Xu, Congying; The Hong Kong University of Science and Technology, Computer Science and Engineerin Terragni, Valerio; The University of Auckland Zhu, Hengcheng; The Hong Kong University of Science and Technology, CSE Wu, Jiarong; The Hong Kong University of Science and Technology, CSE Cheung, Shing-Chi; The Hong Kong University of Science and Technology, Department of Computer Science and Engineering
Computing Classification Systems:	Software testing and debugging

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 functions.

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 MR*), 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 MR-Scout codified MRs are of high quality for automated test case generation, demonstrating the practical applicability of MR-Scout. Furthermore, test cases constructed from the codified MRs can effectively improve test coverage, leading to 13.52% and 9.42% increases in line coverage and mutation score, when compared to developer-written test suites. Additionally, 55.76% to 75.00% of codified MRs can be easily comprehended.

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

1 INTRODUCTION

In recent years, automated test input generation has achieved significant advances [7, 16, 22, 35]. However, constructing test oracles is still a major obstacle to automated test case generation. **Metamorphic Testing (MT)** [9] has been applied to various domains as a promising approach to addressing the test oracle problem [40]. 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 **Metamorphic Relation (MR)**) between these inputs and their corresponding outputs is violated. For instance, determining the expected output of $sin(1)$ is difficult, while a correct implementation of the sine function should provide the same output for $sin(1)$ and $sin(\pi - 1)$. Validating whether the outputs of $sin(1)$ and $sin(\pi - 1)$ are equal becomes much simpler (the associated MR is $sin(x) = sin(\pi - x)$). 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 [40]. However, the design of MR is challenging because it requires domain-specific knowledge and relies on the expertise of testers [1]. This obstructs wider adoption of MT [40].

Earlier studies have been conducted to systematically identify MRs, such as identifying MRs from software specifications [11, 42] or based on defined patterns [41, 53]. However, these approaches suffer from *a low degree of automation*, i.e., they heavily rely on manual efforts to identify concrete

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.

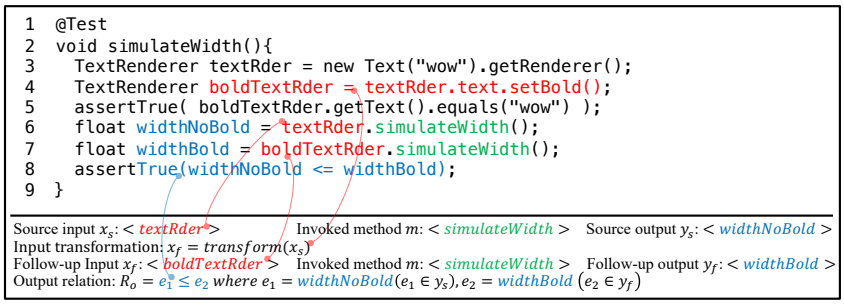


Fig. 1. A test case crafted from `com.itextpdf.layout.renderer.TextRendererTest` in project iText. Underlying MR: *IF* $text_2 = text_1.setBold()$ *THEN* $text_1.width() \leq text_2.width()$.

MRs. On the other hand, several automatic approaches have been proposed to infer MRs for given programs, such as machine-learning-based approaches [5, 28], search-based approaches [51, 52], and genetic-programming-based approaches [3, 4]. However, *wide adoption of these approaches is challenging*. They are mostly designed for the programs in specific domains (e.g., numerical programs [51, 52]) whose input and output values exhibit certain types of relations (e.g., equivalence relations [5], polynomial relations [51], or relations that follow pre-defined patterns [3]).

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* (MTC). 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 original 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* [41]. 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 (such as source and follow-up inputs and the output relation) of encoded MRs. For the `simulateWidth()` case in Figure 1, there is no documentation of the encoded MR relation 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 *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 the

executions of target programs on at least two inputs, (ii) containing the validation of the 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 as *codified MR* (Section 3.2, *MR Synthesis*). Finally, since codified MRs inapplicable to new test inputs are useless for new test generation, MR-Scout filters out codified MRs that demonstrate poor quality in applying to new test inputs (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 soundness of MR-Scout in discovering MTCs, we manually examined 164 samples, and found 97% of them are true positives. This indicates the high precision of MR-Scout in discovering MTCs and the high quality 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 prepared a set of new test inputs for each codified MR. Experimental results show that 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 (Section 4.3, RQ2). Furthermore, to demonstrate the usefulness of synthesized MRs in enhancing test adequacy, we compared test suites constructed from MR-Scout 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 (RQ3, Section 4.4). To evaluate the comprehensibility of codified MRs, we conducted a qualitative study involving three participants and 52 samples. Results show that 55.76% to 75.00% 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 soundness of MR-Scout in discovering MTCs, and the quality, usefulness, and comprehensibility of MRs synthesized by MR-Scout.
- We release the research artifact of MR-Scout and all experimental datasets in our work 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 $\langle f(x_1), \cdots, f(x_n) \rangle$ [10]. Following the definition by Segura et al. [39], an MR can be formulated as

a logical implication from an **input relation** $\mathcal{R}_i$ to an **output relation** $\mathcal{R}_o$.

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

$\mathcal{R}_i$ 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 $\mathcal{R}_i$ is to allow 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 $\langle f(x_1), \cdots, f(x_k)\rangle$. The MR formulation is a general form of that proposed by Chen et al. [9]. It expresses an MR in terms of an input relation and an output relation.

Example 2.1. For example, the property of the sine function $sin(x) = sin(\pi - x)$ can be formulated as an MR: If two inputs $\langle x_1, x_2\rangle$ have the relation $x_2 = \pi - x_1$ ($\mathcal{R}_i$), their corresponding outputs $\langle \sin(x_1), \sin(x_2)\rangle$ should satisfy the relation $\sin(x_1) = \sin(x_2)$ ($\mathcal{R}_o$).

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 [39].

Intuitively, assuming a program implemented by a sequence of statements, MT entails the following five steps [10]: (i) constructing a source input, which can be written by developers or automatically generated (e.g., random testing) [40], (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, 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 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.

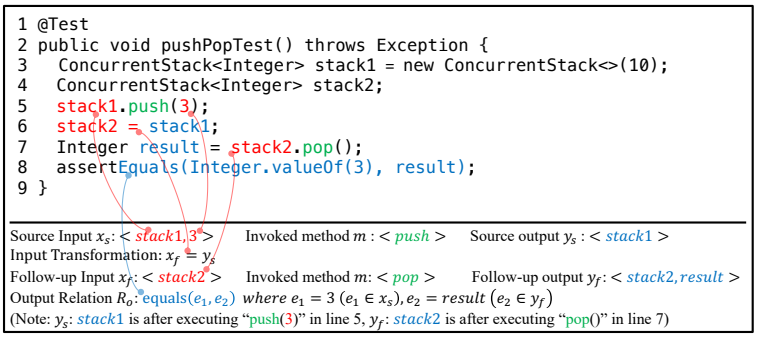


Fig. 2. A test case crafted from `com.conversantmedia.util.concurrent.ConcurrentStackTest` in project Disruptor. Underlying MR: $x = stack.push(x).pop()$ — *IF* an element x is pushed onto a stack and the stack subsequently pops off the top element, *THEN* the element x should be the one popped.

```
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.)

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

For ease of presentation, in the remainder of the paper, we use $m(x)$ to denote $fc(m, x)$, 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 formulated as: *IF* two inputs $\langle x_1, x_2\rangle$ have the relation $x_2.receObj = push(x_1)$ ($\mathcal{R}_i$) , *THEN* the output relation $pop(x_2) = x_1.arg$ ($\mathcal{R}_o$) is expected to be satisfied.

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

Example 2.3. 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 formulated 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 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`, respectively. Arguments are not needed for `simulateWidth()` (i.e., $x_1.arg = x_2.arg = null$).

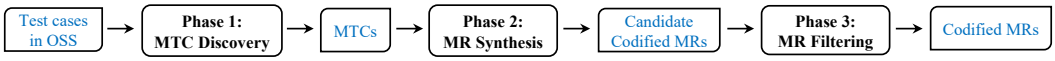


Fig. 3. Overview of MR-Scout

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`, `assertEqual`, `failNotEqual`, `assertArrayEquals`, `assertThat`, `assertIterableEquals`, `assertLinesMatch`	`assertEqual(`e_1, e_2`);` `assertEqual(Math.abs(`e_2`), Math.abs(`e_1`));`

`assertTrue(widthNoBold <= widthBold)` validates if the execution results of `textRder.simulateWidth()` and `boldTextRder.simulateWidth()` satisfies the expected output relation $\mathcal{R}_o$.

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 the test cases collected from open-source projects and returns the 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). Since an MR is defined over at least two inputs and corresponding outputs. These two properties guarantee the executions 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 (MTC) 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, follow-up input) and then codifies their constituents into parameterized methods to facilitate automated test case generation. Such methods are termed as *codified MRs* in this paper (Section 3.2).
(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 [38]. 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 which can be mechanically analyzed. Then, MR-Scout checks these properties in given

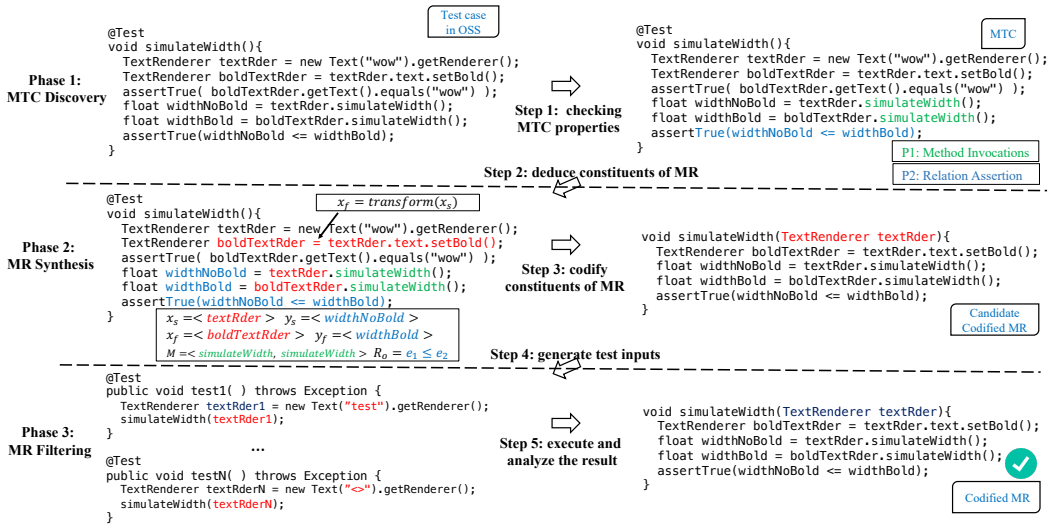


Fig. 4. Procedure of MR-Scout operating on MTC `simulateWidth()`

test cases from open-source projects. Test cases that satisfies 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: $|MI| \geq 2$. *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 *CUT*, 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 executions, it indicates that at least two inputs and corresponding outputs can exist. Specifically, we allow the invocations to the same or different methods of *CUT*.

P2 Relation Assertion: $|RA| \geq 1$. *The test case should contain at least one assertion checking the relation between the inputs and outputs of the invocations in MI.* *RA* denotes the set of relation assertions. 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 method invocations of internal classes that are native to the project under analysis. Excluding 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 [54]. For each internal class with at least two method invocations, the class is considered as a class under test (*CUT*). All classes under test and corresponding method invocations are collected (denoted as $\langle\langle CUT_1, MI_1\rangle, ..., \langle CUT_m, MI_m\rangle\rangle$,) to facilitate *P2-Relation Assertion* identification. If $m \geq 1$, it indicates that test case τ satisfy *P1-Method Invocations*.

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.

(1) Elements of An Output Relation. According to the formulation of MR, an output relation is defined over a set of inputs and outputs. Additionally, there are two necessary elements of an output relation. One is the source input or the source output, and the other is the follow-up output. For a *CUT*, we denote the corresponding method invocations $MI=\{mi_i\}_{i=1}^{n}$, where each mi_i consists of a method being invoked m_i, a set of input x_i, and a set of output y_i (i.e., $mi_i = \langle m_i, x_i, y_i \rangle$). Let E_α denote the elements in an assertion α ($E_\alpha \subseteq \bigcup_{i=1}^{n}(x_i \cup y_i)$). If α is verifying an output relation, E_α should satisfy the constraint:

$$\exists e_1, e_2 \in E_\alpha \text{ s.t. } e_1 \in x_i \cup y_i \wedge e_2 \in y_j \wedge i < j.$$

$i < j$ indicates that mi_j is invoked after mi_i. This constraint allows e_1 to be the source input or output and e_2 to be the follow-up output.

Next, we discuss what are the input set x_i and output set y_i of a method invocation mi_i (where $mi_i = \langle m_i, x_i, y_i \rangle$). By the specification of Java, 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.

- x_i includes (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).
- y_i includes (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).

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 ($e_1 \in x_1$), e_2 =`result` ($e_2 \in y_2$) and E_α = {3, `result`}.

(2) Patterns of Relation Assertions. In addition to the above constraint telling if an assertion includes necessary elements ($e_1, e_2 \in E_\alpha$) 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 an existing work on synthesizing assertion oracles with a set of boolean and numerical operators [44], the principle of two patterns is that an output relation assertion should be a boolean expression where necessary elements e_1 and e_2 are related by (i) numerical operators or user-defined boolean methods (*A1-BoolAssert*) or (ii) assertion methods provided by testing frameworks (*A2-CompAssert*).

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

Example 3.1. For the assertion in line 8 in Figure1, the assertion `assertTrue(widthNoBold <= widthBold)` can be mapped onto *A1-BoolAssert*. `widthNoBold` (e_1) and `widthBold` (e_2). are related by an 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 3.2. For the test case shown in Figure2, the assertion `assertEquals(Integer.valueOf(3), result)` can be mapped onto *A2-CompAssert*, where e_1 =`3` and e_2 =`result`. e_1 and e_2 are related by method `Arrays.equals` which returns a boolean value.

The above two patterns can cover the most commonly used assertion APIs. In Table 1, the corresponding APIs in JUnit4 [25] and JUnit5 [27] 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, with no 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 test case τ, assertions fitting into the above two patterns are considered relation assertions (*RA*). If $RA|\geq 1$, it indicates that *P2-Relation Assertion* is satisfied in test case τ, and τ is discover 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 τ to facilitate the following MR synthesis.

3.2 Phase 2: MR Synthesis

With discovered MTCs in Phase 1, MR-Scout synthesizes MRs from these MTCs in this phase. 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. Currently, 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*. The purpose of synthesizing MRs as methods parameterized with source inputs is to facilitate automated test case generation by integrating with automatic test inputs generation techniques (e.g., random testing [16, 35]). These codified MRs

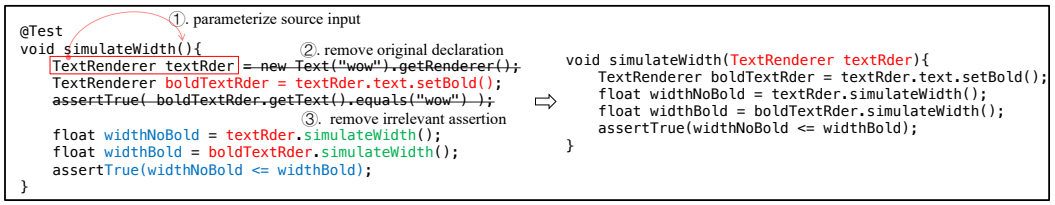


Fig. 5. Illustration of constructing a codified MR

are composed of (i) the input transformation, (ii) the execution of source and follow-up inputs, and (iii) the output relation assertion. By making these methods parameterized with source inputs, new values of source inputs can be easily generated by automatic tools (e.g., Randoop [35] and EvoSuite [16]) and utilized for automated test case generation.

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. For each MR instance identified in an MTC from Phase 1, MR-Scout deduces a tuple of detailed constituents, denoted as $\langle M, x_s, x_f, transform, y_s, y_f, \alpha\rangle$. These constituents include the target methods M, the source input x_s and follow-up input x_f, the input transformation *transform* which makes $x_f = transform(s)(s \subseteq x_s \cup y_s)$, the source output y_s, follow-up output y_f and the output relation assertion α.

Follow the notations in Phase 1 (Section 3.1.2), given an identified MR instance $\langle \alpha, MI\rangle$ (where $MI=\{mi_i\}_{i=1}^{n}$, $mi_i = \langle m_i, x_i, y_i\rangle$), the deduction is as follows.

(1) M are methods invoked in MI.

(2) $\mathcal{R}_i$**-related:** x_s, x_f, and *transform*. MR-Scout first identifies the existence of transformation $x_2 = transform(s)$ $(s \subseteq x_1 \cup y_1 \wedge x_1, y_1 \in mi_1 \wedge x_2 \in mi_2 \wedge mi_1, mi_2 \in MI)$, and then takes x_1 as x_s and takes x_2 as x_f.
Note that not all MR instances have the input transformation because x_f can be hard coded rather than constructed from x_s and y_s. This paper only focuses on MRs with input transformation. Besides, we synthesize MRs from MR instances that involve exactly two method invocations ($|MI| = 2$) which is commonly used [10, 11, 40, 53]. Our evaluation results reveal that 64.13% of MR instances only involve two method invocations (Section 4.1), indicating 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) $\mathcal{R}_o$**-related:** y_s, y_f, and α. MR-Scout directly takes the x_s corresponding output as y_s and the x_f corresponding output as y_f, and takes the output relation assertion α in the identified MR instance.

Example 3.3. As to the example in Figure 4 (Phase 2), there is only one MR instance $\langle \alpha, MI\rangle$ where α= `assertTrue(widthNoBold <= widthBold)`, MI= ⟨`textRder.simulateWidth()`, `boldTextRder.simulateWidth()`⟩.
The identified constituents $\langle M, x_s, x_f, transform, y_s, y_f, \alpha\rangle$ are M:`simulateWidth(), simulateWidth()`, x_s: `textRder`, x_f:`boldTextRder`, *transform*:`boldTextRder = textRder. text.setBold()`, y_s: `widthNoBold`, y_f:`widthBold`, and α: `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 original 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 are 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.

These codified MRs encompass steps 2-5 of metamorphic testing, which involve 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 for the purpose of generating 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 [38]. Therefore, in this phase, MR-Scout filters out low-quality codified MRs, i.e., 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 [52], 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 infers 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, i.e., satisfying the input constraints. An invalid input might not trigger an exception due to the lack of 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. Such cases may produce false alarms and filter out high-quality MRs wrongly.

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 [35], search based [15, 22], and symbolic execution based techniques [7]. Following existing works on test oracle assessment and improvement [24, 44], MR-Scout employs EvoSuite [15] to generate new inputs for codified MRs. Different from Zhang et al.'s work [52] 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 [19, 30], for each codified MR, we ran EvoSuite 10 times with different seeds and gave a time budget of 2 minutes for each run. The detailed configuration of EvoSuite can be found on MR-Scout's website [47].

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 which can apply to at least 95% of generated valid inputs.

4 EVALUATION

Our evaluation aims to answer the following research questions:

RQ1 Soundness: 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)

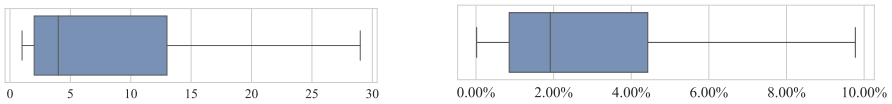
RQ1 aims to evaluate the soundness of MR-Scout in discovering MTCs, i.e., whether discovered test cases possess the defined properties of an MTC. To answer RQ1, we ran MR-Scout on 4,068 OSS projects and built a dataset of 11,350 MTCs discovered by MR-Scout. Then, we manually analyzed 164 sampled MTCs. **RQ2** aims to evaluate the quality of MR-Scout codified MRs, using a set of new test inputs not present in the filtering phase of methodology. 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 of MTC in OSS

We selected open-source projects from GitHub.com [20]. 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, contemporary Java projects which are more likely to use mature unit testing frameworks like JUnit [26] and TestNG [45]. The number of stars indicates the popularity and correlates with project quality [6]. We considered projects created after 01-January-2015 to exclude old projects that might require obsolete dependencies and frameworks. By 05-April-2022, 7,395 repositories met these criteria and were collected. We cloned the latest version of each 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 3327 repositories without tests. At last, we had 4,068 projects, which contain 1,021,129 Java tests from 545,886 test files. These projects comprised 239,724,897 lines of production code and 80,130,804 lines of test code.

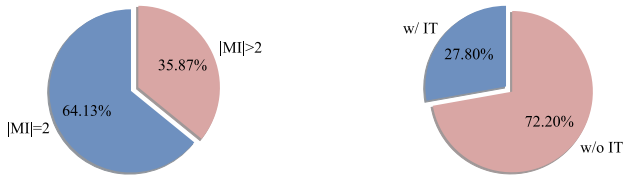
We ran MR-Scout on each of the 4,068 projects on a machine with dual Intel® Xeon™ E5-2683 v4 CPUs and 256 GB system memory. The analysis took 18 hours and 58 minutes, with an average analysis time of 16.78 seconds per project. Finally, MR-Scout discovered 11,350 MTCs in 701 (17.23%) projects. On average, each project has 16.19 MTCs.

Distribution of MTC. 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



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

Fig. 6. Distribution of 11,350 MTCs in 701 projects w.r.t the number and percentage



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

Fig. 7. Percentage of MR instances involving two method invocations (|MI|=2) and input transformation (IT)

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 was 1.91%. Half of the 701 projects have MTC percentages between 0.8% and 4.42%.

We also examined the top 25 projects with the highest number of MTCs (the projects can be found at [47]). 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 (64.12%) out of the 21,574 MR instances involved only two method invocations. Among these 13,836 MR instances, 3,847 (27.80%) in 2,743 MTCs were associated with an input transformation. This indicates that a significant proportion (64.12%) of MR instances leverage MRs involving only two method invocations, and 72.80% 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 widely spread across open-source projects of different domains. 17.23% of projects contain MTCs, and totally 11,350 MTCs were discovered from 701 projects. Besides, the majority of encoded MR instances (64.12%) involve relations with two method invocations. The MTC dataset is released and available on MR-Scout's site [47].

4.2 RQ1: Soundness

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 sound in discovering MR-encoded test cases in real-world projects. The results also reflect the quality of our released dataset of discovered MTCs.

```
...
    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

To achieve this goal, we manually validate if the MTCs discovered by MR-Scout possess 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 sampled 164 MTCs for manual validation. Such a sample size ensures a confidence level of 99% and a confidence interval of 10% for our analysis result.

During the validation, two authors of this paper independently inspect the source code of the discovered test cases. Based on their understanding, they labeled the test cases with one of the following labels:

- *True Positive* indicates the test case possesses the two properties mentioned in Section 3.1, or
- *False Positive* indicates the test case does not possess the two properties mentioned in Section 3.1, or
- *Unclear* indicates the author cannot understand the test case or tell if the two properties are possessed.

After independent labeling, the two authors discussed the 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 positive, whereas the remaining four were labeled as false positive. Overall, MR-Scout achieves a precision of 97% 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 fulfills *P2-Relation Assertion*— validating the relation over outputs of `CUT.abs(x)` and `CUT.abs(x*x)`. Consequently, MR-Scout falsely considers this test case to be positive. However, before assertion, the variables `m` and `n` may be re-assigned with the return values of `min(x,x*x)` and `max(x,x*x)` which are not methods in the class under test. `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 quality.

Answer to RQ1: MR-Scout is sound in discovering MTCs in real-world projects, achieving a precision of 97% in discovering MTCs. Such high precision ensures that our released dataset of MTCs is of high quality.

4.3 RQ2: Quality

4.3.1 Experiment Setup. In Phase 3 *MR Filtering* of the methodology, MR-Scout filters out low-quality MRs with new inputs generated by EvoSuite within a certain time budget. For each MR, MR-Scout runs EvoSuite 10 times with different seeds, allotting 2 minutes for each run. This approach differs from Zhang et al.'s work [52], 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 use the criterion from previous work [52], which considers *an MR that can apply to at least 95% of valid inputs as a* ***high-quality MR***. 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.

Codified MRs Preparation. 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, within MR-Scout discovered MTCs, we collected 485 MTCs from 104 projects.

With collected MTCs, in Phase 2: *MR Synthesis* of MR-Scout (Section 3.2), encoded MRs from 441 (90.92%) MTCs were successfully synthesized into candidate codified MRs which were successfully compiled. The other 9.08% of MTCs were unsuccessfully codified due to too complicated external dependencies or code structures. In Phase 3: *MR Filtering* of MR-Scout (Section 3.3), MR-Scout filters candidate codified MRs with inputs generated by EvoSuite. EvoSuite can successfully generate valid inputs for 125 candidate codified MRs. The main reasons why some codified MRs do not have generated valid inputs include too complex preconditions, incompatible environment, violation of input constraint, etc., which are detailly discussed in Section 5. Among 125 candidate codified MRs that have valid inputs, 60.00% (75/125) of codified MRs pass the *MR Filtering* phase and are finally outputted by MR-Scout as high-quality codified MRs. In this RQ, we evaluate the quality of these codified MRs.

Valid Inputs Generation. In line with the configuration of the previous studies [19, 30], 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[1]. 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, 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. 4 codified MRs did not have newly generated valid inputs. Figure 8 shows the distribution of generated test inputs.

4.3.2 Result. Out of 71 codified MRs with valid inputs generated by EvoSuite, 97.18% (69) are high-quality and even apply to all valid inputs. 2 codified MRs are low-quality, whose 16 (out of 24) valid inputs result in AssertError alarms. After manually analyzing, we conclude 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

[1]The detailed configuration of EvoSuite can be found on MR-Scout's website [47].

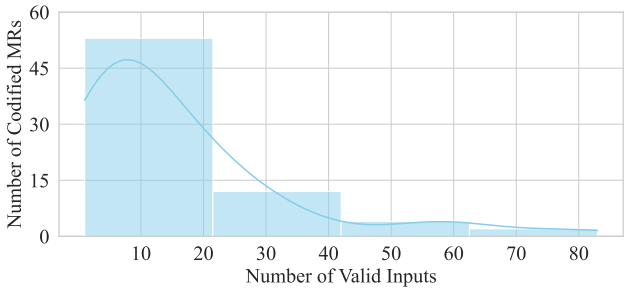


Fig. 8. Distribution of generated valid inputs

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. One application scenario of MR-Scout synthesized MRs is testing the original programs where these MRs are found. In this RQ, we aim to evaluate the usefulness of codified MRs in enhancing the test adequacy of original programs.

Metrics and Baselines. We employ the following four metrics to measure the test adequacy.

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

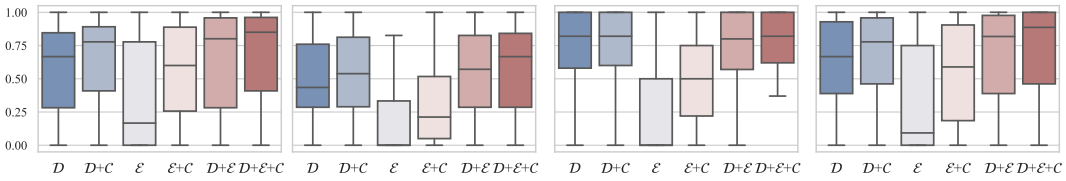
To make use of codified MRs for automated test case generation, firstly, codified MRs are integrated with automatically generated valid inputs in the **RQ2** (Quality) to construct codified-MR-based test suites ($\mathcal{C}$). Then, we compare the performance of the codified-MR-based test suite on these 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. 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 [37], a mutation testing tool, to generate 2,170 mutants for 51 target classes of 75 codified MRs (which are the same codified MRs in RQ2). There are a total of 4,701 lines of code in these target classes.

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 13.52% increase in the line coverage, and 13.37% and 9.42% increase 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 $\mathcal{E}$ ($\mathcal{D}$+$\mathcal{E}$), 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

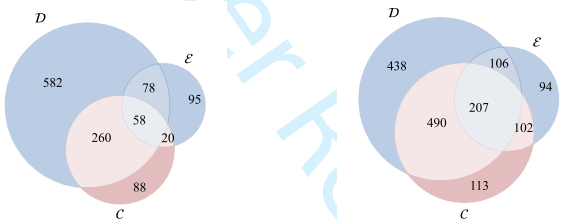
Table 2. Enhancement of test adequacy by codified-MR-based test suites (C) on top of developer-written ($\mathcal{D}$) and EvoSuite-generated test suites ($\mathcal{E}$)

Metrics	VS. $\mathcal{D}$			VS. $\mathcal{E}$			VS. $\mathcal{D}$+$\mathcal{E}$		
	$\mathcal{D}$	$\mathcal{D}$+C	Enhancement	$\mathcal{E}$	$\mathcal{E}$+C	Enhancement	$\mathcal{D}$+$\mathcal{E}$	$\mathcal{D}$+$\mathcal{E}$+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%
P. 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) P. of Covered Mutants

Fig. 9. Enhancement of test adequacy by codified-MR-based test suites (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 (C) test suites

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 (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 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 presents the mutants covered and killed by $\mathcal{D}$, $\mathcal{E}$, or C. Compared with the total mutants covered (1437) and killed (1093) by $\mathcal{D}$ or $\mathcal{E}$, C has 113 (7.86%) exclusively covered mutants and 88 (8.05%) exclusively killed mutants, highlighting that C can further complement the tests written by developers or generated by EvoSuite. The results indicate that MR-Scout codified MRs are effective in enhancing test adequacy (i.e., line coverage and fault-detection capability) on top of the develop-written test suite and EvoSuite-generated test suite.

The enhanced test adequacy from C results from the effective integration of high-quality test oracles (i.e., codified MRs) with a few test inputs generated by EvoSuite. In $\mathcal{D}$, although test oracles are well-crafted and invaluable, each oracle typically applies to one test input. On the other hand, $\mathcal{E}$ is generated with a large number of test inputs, but falls short in the quality of test oracles [16, 18].

C merges the merits of both $\mathcal{D}$ and $\mathcal{E}$. When compared to $\mathcal{D}$, C leverages the same reliable test oracles but with a greater diversity of random test inputs that explore more branches of the target programs. Compared to $\mathcal{E}$, C may not have more test inputs, but offers better test oracles and more meaningful sequences of method invocations (since codified MRs are structured by at least two method invocations). Nevertheless, EvoSuite's random generation of test inputs and sequences may not succeed in invoking certain methods that require complex pre-conditions to invoke but are invoked in codified MRs. 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 leads to `13.52%` and `9.42%` increases in line coverage and mutation score even compared to developer-written test suites, demonstrating the practical usefulness of MR-Scout synthesized MR in 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. On the other hand, 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 three PhD participants who are experienced in programming in Java and MT. All participants have one to two years 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 are collected in the earlier evaluation RQ2, *Codified MRs Preparation*) for the qualitative study. Statistically, this sample size has a confidence level of 99% and a confidence interval of 10% for our analysis result.

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 [12]: very difficult to understand, difficult to understand, easy to understand, and very easy to understand.

4.5.2 Result. Figure 11 shows the participants' response to the comprehensibility of codified MRs. Overall, 55.76% to 75.00% 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 25.00% to 44.24% of the sampled codified MRs are difficult (or very difficult) to understand.

After participants rated the understandability of MRs, we 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). 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 functions 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

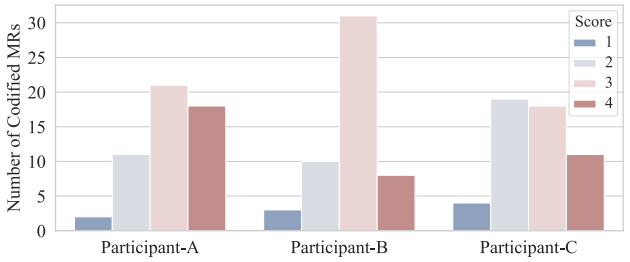


Fig. 11. Comprehensibiliy scores of 52 MR-Scout synthesized MRs (Score: 1. very difficult, 2. difficult, 3. easy 4. every easy to understand)

to understand. Familiarity with the projects would likely mitigate the difficulties posed by class complexity.

Answer to RQ4: 55.76% to 75.00% of codified MRs can be easily comprehended, showcasing the potential of MR-Scout synthesized 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 soundness (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 validate 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 of 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 soundness (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 OSS Projects. Our findings might be influenced by the representativeness of the selected OSS projects. To mitigate this threat and ensure the generalizability of our findings, we rigorously adhered to objective criteria commonly adopted by previous works [23, 49], as described in Section 4.1, to select high-quality and well-maintained Java projects.

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 [19, 30] to run EvoSuite 10 times and choose appropriate parameters that fit our scenario.

5.2 Limitations and Future Work

Despite its effectiveness 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 completeness of their encoded MRs is more decidable 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 employs the static analysis on the source code, and does not well handle the scopes of variables, leading to the false-positive (FP) problem we discussed in Section 4.2. Our sampling result (4 FPs out of 164 samples) reveals that this problem is relatively minor in practice.
(4) MR-Scout filters high-quality codified MRs based on the pass ratio (i.e., at least 95%) of valid inputs. Due to the lack of checks for illegal arguments in methods of classes under test, invalid inputs may reach assertion statements, violate the output relation, and produce false alarms. This may cause some high-quality MRs to be discarded. Effectively 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 [17]: 1) Complex precondition of codified MRs: the time budget may be not enough for EvoSuite to construct complex objects that involve many dependencies or deep hierarchies; 2) Incompatible environment: EvoSuite can be incompatible with some libraries or dependencies in the target project; 3) Bugs of EvoSuite: EvoSuite has bugs that cause crashes during generating inputs for codified MRs; 4) 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); 5) 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 [17], "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

Many studies proposed MRs for testing programs of various domains (e.g, compilers [13, 14, 32, 50], quantum computing [36], AI systems [2, 8, 31, 33, 46, 48]). We review and discuss the most closely related work in systematically identifying MR.

MR Pattern Based Approaches. Segura et al. [41] proposed six MR output patterns for Web APIs, and a methodology for users to identify MRs. Similarly, Zhou et al. [53] 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), and not general, (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 [11], 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+ [42], 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. [38] 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. [52] 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. [51] 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. [3] 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. [43] 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 [28] 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. [29] 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. [5] introduced MeMo, which automatically derives metamorphic equivalence relations from Javadoc, and translates derived MR into oracles. Different from Memo, MR-Scout 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.

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 MR. Our qualitative study shows that 55.76% to 75.00% 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 [47] 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 use Grammarly [21] and ChatGPT [34] to check grammar.

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] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

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

[16] 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

[17] 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

[18] 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

[19] 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

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

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

[22] 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

[23] 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

[24] 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

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

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

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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

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

[35] 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

[36] 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

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

[38] 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

[39] 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

[40] 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

[41] 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

[42] Chang-Ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. 2021. METRIC$ˆ{+}$+: 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

[43] 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

[44] 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

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

[46] 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

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

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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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