MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing

Anonymous Author(s)

ABSTRACT

While a recent study reveals that many developer-written test cases can encode a reusable Metamorphic Relation (MR), over 70% of them directly hard-code the source input and follow-up input in the encoded relation. Such encoded MRs, which do not contain an explicit input transformation to transform the source inputs to corresponding follow-up inputs, cannot be reused with new source inputs to enhance test adequacy.

In this paper, we propose MR-Adopt (*Automatic Deduction Of inPut Transformation*) to automatically deduce the input transformation from the hard-coded source and follow-up inputs, aiming to enable the encoded MRs to be reused with new source inputs. With typically only one pair of source and follow-up inputs available in an MR-encoded test case as the example, we leveraged LLMs to understand the intention of the test case and generate additional examples of source-followup input pairs. This helps to guide the generation of input transformations generalizable to multiple source inputs. Besides, to mitigate the issue that LLMs generate erroneous code, we refine LLM-generated transformations by removing MR-irrelevant code elements with data-flow analysis. Finally, we assess candidate transformations based on encoded output relations and select the best transformation as the result. Evaluation results show that MR-Adopt can generate input transformations applicable to all experimental source inputs for 72.00% of encoded MRs, which is 33.33% more than using vanilla GPT-3.5. By incorporating MR-Adopt-generated input transformations, encoded MR-based test cases can effectively enhance the test adequacy, increasing the line coverage and mutation score by 10.62% and 18.91%, respectively.

KEYWORDS

Software Testing, Metamorphic Testing, Metamorphic Relation, Input Transformation, Code Generation, Large Language Models

1 INTRODUCTION

Metamorphic Testing (MT) is a powerful testing technique to address both the test case generation and the oracle problem [5, 30]. Instead of assessing the outputs of individual inputs, MT validates the behavior of a subject under test (SUT) against a series of Metamorphic Relations (MRs) for the SUT. Each MR defines an *input relation* over a set of related test inputs and an *output relation* over the expected outputs for those inputs. One *outstanding benefit* of MT is that once an MR is identified, MT can leverage a wide range of automatically generated test inputs (known as *source inputs*) to exercise diverse program behaviors with no need to prepare oracles for individual inputs [44]. MT has achieved success in detecting critical faults for various software, such as compilers [17, 34] and databases [22, 25].

Identifying appropriate MRs for a SUT is an essential step to applying MT. As such, there are studies focusing on the MR identification. Earlier approaches either suffer from being labor-intensive and specific to certain domains or pre-defined MR patterns [35, 49, 50] or produce overly generic MRs that are ineffective for testing, as well as recent LLM-based techniques [33, 38]. Recently, Xu et al. [44] report that developers often encode domain knowledge in test cases that exercise MRs. These encoded MRs are found able to be generalized to many new inputs and serve as oracles to test the original programs (or programs with similar functionalities) more exhaustively, by integrating with automatic input generation techniques [5, 30, 44].

However, Xu et al. [44] show that over 70% of 11,000 MR-encoded test cases (MTCs) in their dataset do not contain explicit input relations. Instead, developers often hard-code the source and follow-up inputs. Figure 1a shows an MR-encoded test case intended to have the follow-up input (`dateB`) one day after the source input (`dateA`), but it simply hard-codes the two inputs. Without an explicit input transformation program, follow-up inputs cannot be directly generated from automatically generated source inputs. This limitation hinders the reuse of valuable encoded MRs to achieve automated MT and enhance test adequacy. **This paper aims to overcome this obstacle by inferring an explicit input relation from a given test case with its hard-coded input pairs.** Specifically, our goal is to construct an input transformation function that turns a source input into a follow-up input as shown in Figure 1b. With such input transformations, these encoded MRs can apply to a wider range of test inputs to test SUTs more thoroughly (Figure 1c).

In fact, our goal can be viewed as a programming by example (PBE) task, whose goal is to synthesize an input transformation function that takes the example input (hard-coded source input) and generates the example output (hard-coded follow-up input). It is a non-trivial task as *it requires a correct understanding of the contextual information*, such as the underlying relation between hard-coded input pairs, corresponding output relations, and the properties of SUT. Moreover, in our task, there is only one pair of source and follow-up inputs available as the example [44]. Existing PBE studies suggest that a small number of examples tend to make a generated program overfitted to the given example instead of realizing the true intention [1, 11, 29]. As such, it becomes notably important in our task to *effectively leverage the available contextual information to guide PBE* so that we can generate a *generalizable* input transformation that matches the semantic of an encoded MR, i.e., a generated transformation can apply to all potential source inputs of this MR with its output relation.

In this paper, we propose MR-Adopt, an approach to automatically generating input transformation functions for MRs encoded in human-written test cases leveraging large language models (LLMs). LLMs are trained on extensive code corpus encompassing a variety of programs and tests from various domains and have shown effectiveness in code understanding [10, 24, 26] and generation [3, 7, 15]. Thus, LLMs have the potential to understand the contextual information and generate code based on such information. Our insight is to leverage the code understanding ability of LLMs to mine the intention of MR and input relation from the hard-coded test inputs and SUT's function, and take advantage of their code generation ability to produce good input transformation code. Specifically, we propose three designs to effectively make use of LLMs' abilities.

Firstly, we observe that directly providing LLMs with contextual information only results in around 50% generalizable transformations (Section 4.5). This is unsatisfactory. We need a pipeline that allows LLMs to effectively express the input relation inferred from the hard-coded inputs and generate transformation code. *To realize this goal,* we design MR-Adopt with two phases. In *Phase1*, we prompt LLMs to do analogical reasoning [40, 47] on the hard-coded source-followup input pairs to infer new input pairs that obey the same input relation. In *Phase2*, we use LLMs to generate an input transformation function based on (i) the test input pair hard-coded by developers and (ii) additional input pairs generated by LLMs in *Phase1*. This design not only enables LLMs to generate code in their familiar programming setup (where a task description and several examples are provided) [21], but also mitigates the above-mentioned overfitting issue due to the limited number of examples.

Secondly, we found that LLMs often generate task-irrelevant code segments, of which some are even faulty. For example, when we ask LLMs to generate a test input, they may return a code including a wrong assertion statement. *To remove irrelevant code which can be buggy,* MR-Adopt refines LLM-generated codes by conducting data-flow analysis to extract the codes relevant to the given task (i.e., additional input pairs and input transformation functions).

Thirdly, to mitigate the errors in the relevant codes generated by LLMs, we propose to leverage the developer-written output relations (i.e., assertions) in MTCs as oracles to verify the generated test pairs. We further employ additional inputs to identify an input transformation with the best generalizability as the result.

We evaluated MR-Adopt with 100 developer-written test cases that encode MRs. Experimental results show that MR-Adopt can generate compilable input transformations for 95 MRs, where 72 can generalize to all potential source inputs prepared in our evaluation. It generates 17.28% more compilable transformations and 33.33% more generalizable transformations than directly prompting GPT-3.5. Besides, we found that MR-Adopt-generated transformations effectively produce follow-up inputs for 91.21% source inputs prepared in our evaluation, representing a 122.10% improvement over GPT-3.5 in generating corresponding follow-up inputs for given source inputs. In addition, our ablation study suggests that all three designs (i.e., additional input pairs, date-flow analysis based refinement, and output-relation based validation) contribute to MR-Adopt's overall performance, with the validation and additional example input pairs having the most significant impact. Experimental results also indicate that incorporating MRs equipped with input transformations with automatically generated new inputs leads to a 10.62% increase in line coverage and an 18.91% increase in mutation score on top of developer-written test cases. This demonstrates the practical usefulness of MR-Adopt-generated transformations in enhancing test adequacy.

Contribution. Our work makes the following contribution:

- To the best of our knowledge, we are the first to generate input transformations for MRs encoded in test cases. With the generated input transformations, more encoded MRs can be reused to enhance the test adequacy of SUTs.
- We design and implemented MR-Adopt, an LLM-based approach to deducing input transformation function. MR-Adopt uses a two-phase pipeline to instruct LLMs to generate example test input pairs and then transformation functions. MR-Adopt incorporates a code refinement strategy based on data flow analysis and a validation strategy to mitigate the faulty code generated by LLMs.
- We extensively evaluate the effectiveness of MR-Adopt in generating input transformations. Evaluation results show that MR-Adopt can generate effective input transformations, where 72% input transformations are generalizable to all prepared source inputs. Integrated with the generated input transformation, the encoded MRs increase line coverage by 10.62% and mutation score by 18.91%.
- We build a dataset with 100 encoded MRs from projects after 01-April, 2023. We release this dataset and our replication package on our website [37].

2 PRELIMINARIES

2.1 Metamorphic Testing

Metamorphic Testing (MT) validates a program P using Metamorphic Relations (MRs). An MR $\mathcal{R}$ can be expressed as a logical implication from an ***input relation*** $\mathcal{R}_i$ to an ***output relation*** $\mathcal{R}_o$ [5, 30, 44].

$$\mathcal{R} = \langle \mathcal{R}_i, \mathcal{R}_o \rangle, \text{ where } \mathcal{R}_i \left(x_s, x_f\right) \implies \mathcal{R}_o \left(y_s, y_f\right)$$

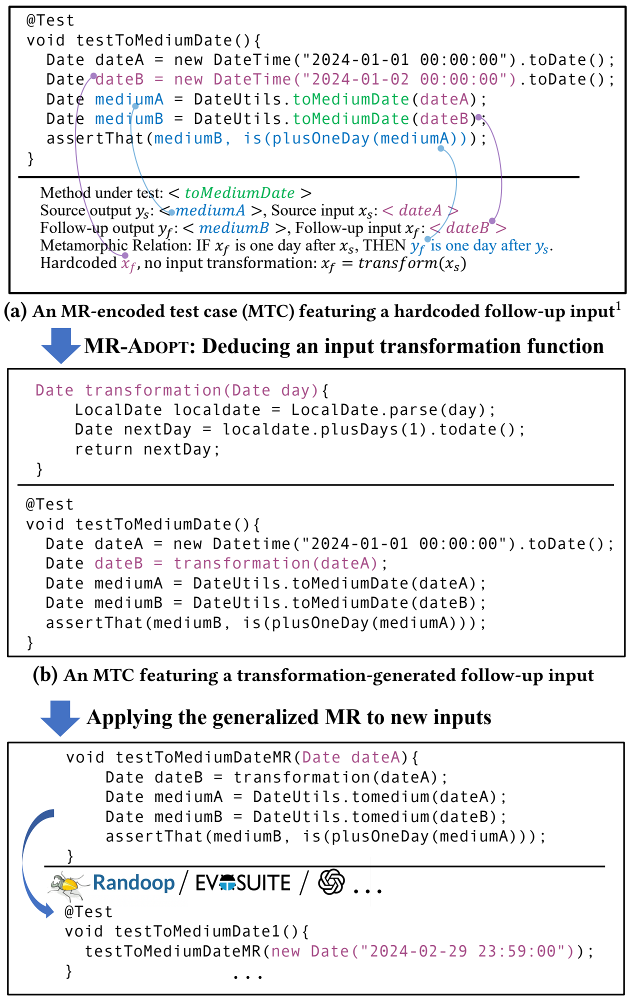
$\mathcal{R}_i$ defines the rule to generate an additional test input (known as the *follow-up input* x_f) from a given test input (known as the *source input* x_s), and $\mathcal{R}_o$ defines the relation between the expected outputs (y_s, y_f) for the source and follow-up inputs, respectively. For example, given a program P implementing the *sine* function, an MR can be defined with the input relation $\mathcal{R}_i$ as $x_f = -x_s$ ($\forall x_s \in \mathbb{R}$) and the output relation $\mathcal{R}_o$ as $y_f = -y_s$. This MR is based on the property that $P(x) = -P(-x)$ should hold for a correctly implemented *sine* function.

Given an MR $\mathcal{R}$ for a SUT P, conducting MT for P entails the following five steps: (i) constructing a source input x_s, (ii) executing P on x_s and obtaining the source output y_s, (iii) constructing a follow-up input x_f that satisfies $\mathcal{R}_i$, (iv) executing P on x_f and obtaining the follow-up output y_f, and (v) verifying if the two outputs y_s and y_f satisfy the output relation $\mathcal{R}_o$. Typically, a function referred to as the ***input transformation*** is designed to implement $\mathcal{R}_i$ to generate x_f from the given x_s, and x_s can be written by developers or automatically generated (e.g., random testing) [30, 44].

2.2 MR-Encoded Test Cases

MR-encoded test cases (MTCs), introduced by Xu et al. [44], refer to the test cases whose embedded domain-specific knowledge suggests useful MRs. Such MTCs are prevalent. In their study, over 11,000 MTCs were discovered from 701 open-source projects. An MTC

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



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

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

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

MR-Adopt: Deducing an input transformation function

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

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

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

Applying the generalized MR to new inputs

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

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

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

Figure 1: Overview of MR-Adopt for Metamorphic Testing

can be considered as an instance of an MR, already implemented with specific source and follow-up input values, invocations of methods under test, and output relation assertions. Such encoded MRs can be generalized to new inputs and facilitate automated MT by incorporating automatic input generation techniques.

Consider the example in Figure 1. The encoded MR in this test case is: "IF a date x_1 in long format ("yyyy-mm-dd hh:mm:ss") is one day ahead of another long-format date x_2 ($\mathcal{R}_i$), THEN x_1 in medium format ("yyyy-mm-dd") should also be one day ahead of medium-format x_2 ($\mathcal{R}_o$)". The SUT method `toMediumDate` is executed on the source input `dateA` and the follow-up input `dateB` separately, and the corresponding source and follow-up outputs are verified by the assertion code `assertThat(mediumB, is(plusOneDay(mediumA))`, which implements $\mathcal{R}_o$.

Such an implemented MR instance can be reused and generalized to many new inputs. However, the follow-up input `dateB` is hardcoded with value `"2024-01-02 00:00:00"` instead of being generated from `dateA` by an input transformation program. That is, even if $\mathcal{R}_o$ is explicitly coded, $\mathcal{R}_i$ is **implicit** behind the specific source and follow-up input values `dateA` and `dateB`. According to Xu et al.'s study, over 70% of MR-encoded test cases lack explicitly coded $\mathcal{R}_i$ (i.e., input transformations). This limitation prevents these MRs from being directly applied to new inputs automatically generated by existing tools, e.g., Evosuite [9] and Randoop [28]. While these tools are proficient in generating diverse source inputs, they struggle with generating input pairs that satisfy an input relation.

In this paper, we aim to address this limitation by deriving an explicit input relation from a given test case and its hardcoded input pairs. Specifically, our goal is to construct an **input transformation function** that converts a source input into a follow-up input, as shown in Figure 1b. With such input transformations, embedded MRs can be reused with a broader range of test inputs (Figure 1c) to exercise more SUT's behaviors, thereby enhancing the test adequacy. One benefit of reusing the encoded MRs to prepare tests is that we reuse the oracles (output relation assertions) written by the developers in the MTCs, which could be fairly reliable.

3 MR-ADOPT

Figure 2 presents an overview of MR-Adopt. It takes as input a pair of source and follow-up inputs and its context (i.e., an MR-encoded test case and methods under test) and outputs an input transformation function.

MR-Adopt works in a two-phase pipeline. In the first phase, it generates additional source-follow-up input pairs and uses them as examples to better describe the input relation, which provides useful guidance for the generation of input transformations. In the second phase, it generates input transformation functions based on these example pairs. This setup is familiar to LLMs for code generation tasks, as it includes not only a task description but also several examples [21]. This two-phase pipeline provides more information to effectively guide LLMs in generating generalized transformations.

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

3.1 *Phase 1:* Input Pair Preparation

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

Following the idea of the Chain of Thought strategy [41], MR-Adopt prompts an LLM in two steps. The LLM is first asked to generate source inputs, and then generate the corresponding follow-up inputs for previously generated source inputs. We adopted this

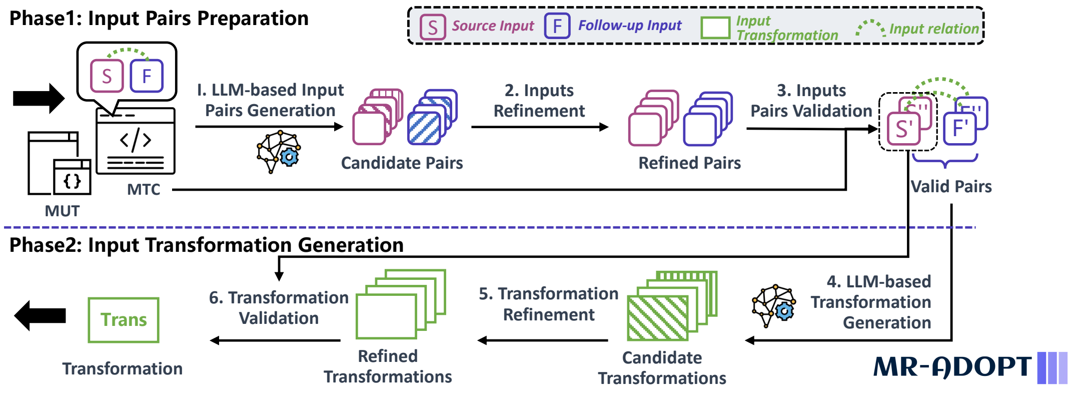

Phase1: Input Pairs Preparation
S Source Input
F Follow-up Input
Input Transformation
Input relation
S
F
MUT
MTC
1. LLM-based Input Pairs Generation
Candidate Pairs
2. Inputs Refinement
Refined Pairs
3. Inputs Pairs Validation
S'
F'
Valid Pairs
Phase2: Input Transformation Generation
4. LLM-based Transformation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

Figure 2: An overview of MR-Adopt.

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

Listing 1: Examples of LLM-generated source inputs

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

Listing 2: Examples of LLM-generated input pairs

step-by-step strategy because we found that LLMs perform better when generating source and follow-up inputs sequentially rather than generating entire input pairs at once.

We design the source input generation prompt following the recent practice [46, 48]. Our prompt includes (i) a system message about the role of a Java expert and the task to generate test inputs, (ii) the code of methods under tests, (iii) the code of the MR-encoded test case, and (iv) the output format. Such a prompt provides necessary contextual information (ii and iii) and task description (i and iv) to generate source inputs. Due to the page limit, the detailed prompt templates and examples can be found on MR-Adopt's website [37]. Listing 1 shows several example source inputs generated by GPT-3.5 with this prompt.

The follow-up input generation prompt is similar to the prompt for source input generation. The difference is that we add the five previously generated example source inputs to the prompt since the current task is to prepare follow-up inputs for the example source inputs. We also modify the task description and output format to ask LLMs to prepare the source-followup input pairs, with the original input pair in the MTC as the sample.

Following recent studies' nucleus sampling [7, 13], MR-Adopt repeats the above generation process five times with a temperature setting of 0.2 [2, 4]. Listing 2 shows several example input pairs.

3.1.2 Input Pair Refinement. Similar to the observations in existing studies [8], we found that LLM-generated codes often include irrelevant segments, which may introduce unnecessary errors. For instance, the second input pair in Listing 2 includes an assertion statement `assertEquals(dateTimeB, is(plusOneDay(dateTimeA)));` alongside the desired input pair code. This assertion is irrelevant to constructing `dateA` or `dateB` and incorrectly calls the method `plusOneDay(Date date)` with a `DateTime` object, resulting in a type mismatch exception.

Our task focuses on constructing source and follow-up inputs. To exclude irrelevant code and bypass unnecessary errors, we perform a data-flow analysis on the code returned by LLMs and build a dependency graph . Then, MR-Adopt identifies the dependent statements of the source and follow-up inputs and removes the other statements. For example, in the second input pair of Listing 2, the source input `dateA` and follow-up input `dateB` depend on objects `dateTimeA` and `dateTimeB`, respectively. Thus, the statements (Lines 11-14) for constructing `dateA`, `dateTimeA`, `dateB`, and `dateTimeB` are considered relevant, while the assertion statement (Line 15) is excluded. Finally, MR-Adopt retains only the statements relevant to constructing source and follow-up inputs, excluding all other irrelevant statements from the LLM-generated code.

3.1.3 Input Pair Validation. The previous refinement step removes the irrelevant code segments generated by LLMs and results in candidate source-followup input pairs. However, a pair of inputs still can be *invalid* if they violate the input relation of an encoded MR. For example, the third input pair shown in Listing 2 is an invalid test pair. The input relation of the embedded MR is that "`dateA` is one day ahead of `dateB`", while an LLM yields an input pair of "2024-01-01" and "2025-01-01", which does not follow the one-day-after input relation. Such test case pairs do not match the
```

```java
1  @Test
2  void testToMediumDate(){
3      // LLM-generated new source input
4      Date dateA = new DateTime("2024-12-31 23:59:59").toDate();
5      // LLM-generated corresponding follow-up input
6      Date dateB = new DateTime("2025-01-01 23:59:59").toDate();
7      Date mediumA = DateUtils.toMediumDate(dateA);
8      Date mediumB = DateUtils.toMediumDate(dateB);
9      assertThat(mediumB, is(plusOneDay(mediumA)));
10 }
```

Listing 3: Validating an LLM-generated input pair

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

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

**Listing 4: An example of output format in the prompt**

intention of the desired input transformation and will mislead the generation of transformations. In this step, we propose a method to discard such invalid input pairs.

We use the output relation of an encoded MR to validate LLM-generated input pairs. Specifically, MR-Adopt executes SUT on generated input pairs and checks the outputs against the output relation of an encoded MR. Note that the output relation is an explicit reusable code in the MTC, i.e., the developer-written assertions (Line 7 in Figure 1a). For each input pair, if its outputs of invoking methods under test on the inputs pass the developer-written assertions, MR-Adopt considers it a valid input pair. As shown in Listing 3, if the outputs `mediumA` and `mediumB` pass the assertion (Line 9), the inputs `dateA` and `dateB` are considered valid. This step aims to filter out invalid input pairs generated by LLMs from the example set. It could discard some source-followup input pairs that match the input relation in fact. Factors such as the bugs in a non-regression SUT may lead to false violations and mistaken deletions of these pairs. However, the goal of the first phase is to prepare examples that give more information about the input relation for the second phase. Thus, it does not require *complete* source-followup input pairs.

## 3.2 *Phase 2:* Transformation Generation

*3.2.1 Transformation Generation.* In this step, MR-Adopt asks an LLM to generate candidate input transformation functions for an encoded MR by giving the example source-followup input pairs. The examples include the original hard-coded pair and the additional ones produced in *Phase1*.

Similar to the prompt for input pair generation (Section 3.1.1), the prompt for transformation generation consists of (i) a system message, (ii) the code of methods under test, (iii) example input pairs, (iv) the code of an MTC, and (v) the output format. The difference is that the task changes from generating source-followup input pairs to generating input transformation functions, whose parameter list and return type have been specified. The detailed prompt template and samples are available on MR-Adopt's website [37].

Listing 4 shows the output format specified in the prompt, which defines the skeleton of the input transformation function to generate. It includes the function name, parameter (i.e., source input) types and names, and type of the return value (i.e., follow-up input) [2]. Following recent studies' nucleus sampling [7, 13], for each MR, MR-Adopt instructs an LLM to generate one input transformation function, and repeats the generation process five times with a temperature setting of 0.2 [2, 4]. Finally, five candidate transformation functions can be generated.

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

*3.2.2 Transformation Refinement.* Similar to the situation discussed in Section 3.1.2, LLM-generated transformation functions can contain irrelevant code, some of which can cause errors (e.g., invoking non-existing APIs). To address this issue, MR-Adopt retrieves the data-flow paths that end at the follow-up inputs. The code involved in these paths is considered relevant, while other code is considered irrelevant and excluded. For example, as shown in Listing 5, the follow-up input `nextDay` depends on `localDate`, which further depends on `day`. Therefore, the statements constructing `nextDay` and `localDate` are retained, while irrelevant statements such as `Date dayAfter = day.after(1)` are excluded.

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

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

**Listing 5: An example of LLM-generated transformation**

After excluding irrelevant code, MR-Adopt analyzes and imports dependencies needed by the generated transformation function. MR-Adopt first identifies the dependent classes' names by using a syntax analysis library JavaParser [3]. Then, MR-Adopt retrieves potential classes defined or employed in source and imports those whose names match the dependent classes. In the example in Listing 5, the dependent internal class `LocalDate` will be imported.

*3.2.3 Transformation Assessment.* After refining candidate transformations, MR-Adopt further assesses their quality by applying them to new source inputs to construct new test cases. In this step, MR-Adopt leverages new source inputs generated in *Phase1* (Section 3.1) to assess the generalizability of these candidates and then selects the best one.

Specifically, MR-Adopt uses new source inputs as test inputs and employs developer-written assertions (i.e., output relation assertions) as test oracles. A transformation is considered applicable to a given source input if (a) the input transformation function can successfully generate a corresponding follow-up input without throwing exceptions, and (b) the outputs from executing the methods under test pass the developer-written assertions. Consider the

[2]For MRs with multiple follow-up inputs, the return type is a list of objects.
[3]https://javaparser.org/
```

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

Listing 6: Validating an LLM-generated input transformation with a new source input

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

Listing 7: An example of LLM-generated transformation

example in Listing 6. Given the source input `dateA`, if the follow-up input `dateB` can be successfully generated and the outputs `mediumA` and `mediumB` pass the assertion (Line 9), MR-Adopt considers this candidate transformation applicable to input `dateA`. Conversely, Listing 7 shows a failing transformation that limits the input to January dates. MR-Adopt assesses all candidate transformation functions using both new source inputs (if any) and the original source input. It then selects the most generalizable transformation that are applicable to the most inputs. In the case of a tie, MR-Adopt will return the first generated one as the result.

4 EVALUATION

4.1 Research Questions

Our evaluation aims to answer the following research questions:

- **RQ1:** *How effective is MR-Adopt in generating input transformations?* This RQ compares the quality of the input transformation functions generated by MR-Adopt and baselines to evaluate the effectiveness of MR-Adopt in generating generalizable input transformations for MRs encoded in MTCs.
- **RQ2:** *How effective are MR-Adopt-generated input transformations in constructing follow-up inputs, compared with LLMs?* This RQ investigates whether explicitly generating input transformation functions is beneficial by comparing the quality of follow-up inputs generated by transformation functions and those directly generated by LLMs.
- **RQ3:** *What is the contribution of each component in MR-Adopt for generating input transformations?* This RQ performs an ablation study to reveal how each design contributes to generating input transformations.
- **RQ4:** *How useful are encoded MRs in enhancing test adequacy with the generated input transformations?* With input transformations generated from MR-Adopt, more encoded MRs can be reused by incorporating new inputs to test more behaviors of SUT. This RQ investigates the usefulness of such encoded MRs in improving test adequacy to demonstrate the value of generating input transformation for these encoded MRs.

4.2 Dataset

MR-encoded test cases (MTCs). We followed Xu et al. [44] to collect the open-source GitHub Java projects receiving at least 200 stars to ensure the quality of the code source. Besides, we further excluded the projects created before 01-April 2023 to ensure that the experimental LLMs have not learned the during their training, thereby reducing the potential for data leakage [2] Finally, we collected 2,007 MTCs from qualified projects. From these MTCs, we retained test cases that (i) can be successfully compiled, (ii) can be successfully executed (i.e., passing developer-written assertions), and (iii) contain MRs associated with exactly two method invocations (one for the source input and one for the follow-up input). The third criterion serves to exclude the complex and less-frequent MRs involving multiple groups of inputs [44]. Finally, we obtained 180 MTCs, including 54 with explicit input transformation functions written by developers and 126 without such functions, which follows a distribution consistent with Xu et al.'s finding [44].

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

Next, we manually constructed ground truth input transformation functions for MTCs without developer-written input transformations. Specifically, 64 out of 126 MTCs without input transformations are randomly selected as tasks. For each task, one author of this paper examined the SUT and its underlying MRs and then created a transformation function that must apply to the original source input and should be generalizable to new valid source inputs. Another author reviewed these transformations, and disagreements were discussed and resolved with consensus. This process took approximately 200 human hours. Details of this dataset can be found on MR-Adopt's website [37].

4.3 Environment and Large Language Models

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

The large language models used in our evaluation include GPT-3.5 from OpenAI [4] and three open-source code models: Llama3-8B [5]

[4]https://platform.openai.com/docs/models/
[5]https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

from Meta, Deepseek-coder-7b [6] from DeepSeek, and CodeQwen1.5-7B-Chat [7] from Alibaba. We use these LLMs since they are popular state-of-the-art [8] code models in well-known LLM families and deployable at our machine.

4.4 Source Input Preparation

To evaluate the generated input transformations and LLMs, we need new valid source inputs as a "test set". As introduced in Section 2.1, automatic test input generation techniques (such as Evosuite [9] or Randoop [28]) can be employed to prepare source inputs. However, we found these tools cannot generate any test inputs for many MRs. This is because over 50% of experimental MRs' inputs are user-defined complex objects with complicated preconditions and environments, which are hard to handle by test input generation tools like Evosuite. This aligns with the observation in Xu et al.'s study [44].

Recent studies show that LLMs are good test input generators [46, 48]. Thus, in this study, we employed an LLM, Qwen, to generate new source inputs for evaluating transformations. As a reminder, the LLM used for source input generation is different from those used for generating input pairs (in MR-Adopt's *Phase1*) and transformation functions (in MR-Adopt's *Phase2*), thereby mitigating the circularity issue in the evaluation. We reused the prompt template from MR-Adopt's *Phase1*. Qwen was asked to generate five source inputs at one time, and we repeated the generation process ten times with a temperature setting of 0.2 to produce more source inputs.

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

4.5 RQ1: Effectiveness of MR-Adopt

4.5.1 Experiment Setup. This RQ inspects the effectiveness of MR-Adopt in generating input transformation functions by testing whether they are compilable and how well they generalize to new source inputs.

Baselines. To the best of our knowledge, no approach has been proposed to generate input transformation functions for MRs in different domains. Considering that LLMs are shown to be powerful in code and test generation, we set *directly prompting LLMs* as baselines. Specifically, we directly prompted GPT-3.5-turbo-0125 (shorten as GPT-3.5), Llama3-8B-Instruct (shorten as Llama3), and Deepseek-coder-7b-instruct-v1.5 (shorten as Deepseek) with the same template as that of MR-Adopt (Section 3.2.1). The knowledge cut-off dates of these modelsare September 2021 [9], March 2023 [10], and March 2023 [11], respectively, before the creation of MTCs in our dataset (Section 4.2),

Table 1: Effectiveness of MR-Adopt in generating input transformations for 100 MRs encoded in test cases

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

# n% generalizable: the number of generated input transformations that can apply to at least n% of source inputs.

Configuration of baseline LLMs. Following recent studies [7], we used the nucleus sampling [13] and repeated the generation process five times for each task with a temperature setting of 0.2 [2, 4], and selected the best result for comparison. As a reminder, the configuration of our method is introduced in Section 3.2.1.

Metrics. For this RQ, we introduced two metrics – (i) *# compilable transformations:* the number of generated input transformations that can be compiled successfully, and (ii) *# n% generalizable transformations:* the number of generated input transformations that can apply to at least n% of source inputs prepared in Section 4.4. $n = 0, 75, 100$ represent at least one, upper-quartile, and all source inputs, respectively. We consider a transformation t *applicable to* a source input x_s when t generates a follow-up input x_f for x_s, such that a *correct* SUT does not violate the output relation on the input pair $<x_s, x_f>$.

4.5.2 Result. As shown in Table 1, MR-Adopt effectively produced many compilable input transformation functions that well generalize to prepared source inputs. We found that MR-Adopt works best with GPT-3.5. Specifically, with GPT-3.5 (the last column), MR-Adopt produced compilable transformations for 95 out of 100 MRs. Among them, 72 transformations were effectively applied to *all* prepared source inputs. MR-Adopt also works well with the open-source Llama3 and Deepseek, which generated 68 and 71 100% generalizable transformations, respectively.

Besides, we found that some generated transformations generalize to some but not all source inputs prepared in our experiment. Specifically, with GPT-3.5, 83 out of 95 compilable transformations apply to at least one source input and 81 of them apply to >75% prepared source inputs. Similar situations are found on Llama3 and Deepseek. We considered these transformations generated by MR-Adopt *still useful* to some extent as they successfully prepare some valid input pairs. Furthermore, we analyzed their issues and found they could be settled with a more comprehensive prompt to handle corner cases. The LLM-generated transformations can handle common cases but struggle with edge cases. For example, an ideal transformation should generate a higher version string for any cases (e.g., transforming `"1.0-A1"` to `"1.0-B1"`), while the LLM-generated one parses the version based on "Major.Minor.Revision" convention (e.g., `"1.0.1"`) and fail to handle cases like `"1.0-A1"`.

[6]https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5
[7]https://qwenlm.github.io/blog/codeqwen1.5/
[8]https://evalplus.github.io/leaderboard.html
[9]https://help.openai.com/en/articles/8555514-gpt-3-5-turbo-updates
[10]https://huggingface.co/NotAiLOL/Meta-Llama-3-8B-Instruct
[11]https://github.com/deepseek-ai/DeepSeek-Coder/issues/89

Table 2: Effectiveness of MR-Adopt's transformations in constructing follow-up inputs for 1366 source inputs

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

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

Also, we found that 5, 14, and 11 transformations produced by MR-Adopt with GPT-3.5, Llama3, and Deepseek cannot be compiled, respectively. The main reasons include: (i) the generated transformations invoke non-existing methods to generate the follow-up input and (ii) they invoke APIs inaccessible due to permission restrictions (e.g., private methods). We also found that the compilable yet not generalizable transformations are mainly because *Phase1* failed to generate valid input pairs for these MRs. Then, LLMs generated transformations overfitted to the given input pair.

We also compared the performance of MR-Adopt (5-7 columns) with the baseline of directly prompting LLMs to generate transformations (2-4 columns). We found that MR-Adopt generates more compilable transformations. We attribute this to our design of code refinement and assessment to extract reliable code from LLMs' responses. Moreover, MR-Adopt demonstrates substantial improvements in generating transformations that are >75% and 100% generalizable, with increases ranging from 15.63% to 28.57% and 18.33% to 33.33%, respectively. This confirms the usefulness of our design of preparing more examples to help LLMs generate transformations as well as the helpfulness of MR-Adopt's refinement and selection strategies.

Answer to RQ1: MR-Adopt significantly outperforms the baseline LLMs across all metrics. Compared to directly prompting LLMs, MR-Adopt achieves 17.3%~33.33% improvement in generating 100% generalizable input transformations.

4.6 RQ2: Effectiveness of Input Transformations

4.6.1 Experiment Setup. This RQ examined the quality of follow-up inputs produced by input transformations generated by MR-Adopt. We set LLMs as the baselines because they are off-the-shelf black-box transformations that can generate follow-up inputs given source inputs, as introduced in Section 3.1.1. We also included LLMs enhanced with MR-Adopt's refinement procedure (marked with $^+$) for comparison. This can reflect the effectiveness of MR-Adopt's refinement for input pairs preparation (Section 3.1.2).

Metric. We generated follow-up inputs by feeding the 1,366 prepared source inputs (Section 4.4) to input transformations generated by MR-Adopt and the vanilla LLM baselines. To compare the qualities of the follow-up inputs produced by the MR-Adopt-generated transformations and the baselines, we used the number of *valid* follow-up inputs as the metric. Similar to Section 4.5, we consider a follow-up input x_f *valid* if it and its corresponding source input can pass developer-written output relation assertions.

4.6.2 Result. As shown in Table 2, when built with GPT-3.5, input transformation functions generated by MR-Adopt produced valid follow-up inputs for 1246 out of 1366 (91.22%) source inputs. The high validity rate demonstrated that MR-Adopt contributed to abundant useful source-followup input pairs.

In comparison, three vanilla LLMs only generated valid follow-up inputs for 697 (51.02%), 724 (53.00%), and 597 (43.70%) source inputs, respectively. MR-Adopt surpassed them by 72.10%-108.71%. LLMs enhanced with MR-Adopt's input refinement procedure introduced in Section 3.1.2 (marked with $^+$) worked better than the vanilla LLMs. This confirmed the usefulness of our design to refine the LLM-generated test inputs (Section 3.1.2). Meanwhile, MR-Adopt's transformations still outperformed the enhanced LLMs by generating 61.82% more valid follow-up inputs than Llama3$^+$, 69.06% more than Deepseek$^+$, and 75.99% more than GPT-3.5$^+$. This significant performance gap highlights the effectiveness of MR-Adopt's transformation functions compared to the state-of-the-art LLMs. It also evidenced the usefulness of our idea to codify the input transformation by leveraging the code understanding and generation abilities through the two-phase pipeline and preparation-refinement-validation process.

We also summarized two major limitations of using vanilla LLMs as black-box transformations based on our observation. Firstly, LLMs can generate a follow-up input with a wrong value, which is similar to the case in Listing 2. Another limitation is that LLMs often fail to capture the constraints between multiple arguments of the follow-up input. For instance, consider a method `deserial(data, size)` to deserialize an `ArrayList data` with a given `size`. The `size` should not be greater than the length of `data`. However, LLMs may miss this constraint and generate invalid value for `size`. These issues about value processing could be due to LLMs' limited inference ability. Instead, MR-Adopt asks LLMs to codify the input transformation and uses the code to do calculation and processing, which is recognized as a better way to exert LLMs' abilities [19]. Besides, we argued that using LLMs as transformations can be costly since we need to request LLMs for each source input. Meanwhile, MR-Adopt uses LLMs to generate transformations for once, and there is no need to query LLMs When using the generated transformations.

Answer to RQ2: MR-Adopt's refinement step can effectively enhance follow-up input generation, with up to 18.59% improvement for GPT-3.5. Additionally, MR-Adopt-generated transformations can effectively generate follow-up inputs for 91.21% source inputs, surpassing GPT-3.5$^+$ by 75.99%.

4.7 RQ3: Ablation Study on MR-Adopt

4.7.1 Experiment Setup. We created three variants v_1, v_2, and v_3 of MR-Adopt by ablating three components to analyze the helpfulness of these designs for generating generalizable input transformations. We chose MR-Adopt built with GPT-3.5 which achieves the best result in RQ1 (Section 4.5). The variants included:

v_1**: MR-Adopt w/o additional input pairs.** This variant used only one source-followup input pair hard-coded in an MTC to guide the input transformation generation. It did not use additional input pairs generated in MR-Adopt's *Phase1* (Section 3.1).

v_2**: MR-Adopt w/o refinement step.** This variant disabled the refinement step for generated input transformations in MR-Adopt (Section 3.2.2).

v_3**: MR-Adopt w/o assessment step.** This variant disabled the assessment step for generated transformation functions (Section 3.2.3). Instead, it randomly selected one of the compilable transformation functions as the result.

Table 3: Contribution of each component in MR-Adopt

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

4.7.2 *Result.* As shown in Table 3, removing additional input pairs (v_1) led to a 19.44% decrease in generating 100% generalizable transformations. This indicated that additional input pairs could effectively alleviate the problem of overfitting caused by the limited number of examples in PBE [1, 11, 29], and help MR-Adopt generate more generalizable transformation.

Similarly, disabling refinement steps (v_2) reduced 15.27% input transformations that generalize to 100% prepared inputs. This indicated that some generated transformations have minor issues and can be refined by excluding irrelevant code. Besides, disabling the assessment step (v_3) decreased 22.22% input transformation generalizable to 100% prepared inputs. This indicated that even with additional input pairs and the refinement step, LLMs still cannot generate transformations that well match the input relation and generalize to prepared inputs. The assessment step is necessary to rank the most generalizable function.

Answer to RQ3: All three designs contribute to the effectiveness of MR-Adopt in generating generalizable transformations. The assessment procedure contributes the most, and additional example input pairs contribute similarly.

4.8 RQ4: Usefulness of Input Transformations

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

New Test Cases Construction. We applied generalized MRs incorporating generated transformations to the automatically generated source inputs introduced in Section 4.4 to obtain a set of new test cases (denoted as $\mathcal{M}$). We compare such test cases against two baselines: (i) the developer-written test cases (i.e., MTCs) (denoted as $\mathcal{D}$) and (ii) test cases based on the LLM-generated source and follow-up input pairs (denoted as $\mathcal{L}$). Specifically, we combined the prepared source inputs (Section 4.4) with valid follow-up inputs generated by Llama3$^+$ which performed the best in RQ2 (Section 4.6). Considering generalized MR based test cases and LLM-generated input pairs based test cases are extended from developer-written existing test cases, we followed Xu et al. [44] to analyze the test adequacy improvement on top of developer-written test cases.

Metrics. We used two metrics of test adequacy: (i) Line Coverage: the percentage of lines of code in the target classes executed by test cases, and (ii) Mutation Score: the percentage of mutants killed by test cases.

Mutation Testing: We employed Pitest [12] to conduct mutation testing. Each MR only focused on one or two methods under test in the target class. To include the covered lines or killed mutants in the methods intransitively invoked by MR-involved methods for comparison, we employed Pitest to generate mutants targeting all methods in a target class. Finally, Pitest successfully generated 4,388 mutants for 45 target classes covered by 88 MRs in the dataset (Section 4.2). Pitest failed for the other 12 MRs' classes because of environmental issues (e.g., conflict dependencies).

Table 4: Enhancement of test adequacy from generalized MR based test cases ($\mathcal{M}$) on top of developer-written ($\mathcal{D}$) and LLM-generated input pairs ($\mathcal{L}$) based test cases

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

4.8.2 *Result.* As shown in Table 4, compared to developer-written MTCs ($\mathcal{D}$), incorporating new test cases constructed from generalized MR ($\mathcal{D}$+$\mathcal{M}$) increased the line coverage by 10.62% and the mutation score by 18.91%. This suggested that MR-Adopt could enhance the test adequacy by integrating high-quality test oracles (i.e., output relation of the encoded MR) with a diverse set of potential test input pairs of the MR ($\mathcal{M}$). Although the developer-written test inputs hard-coded in MTCs were carefully crafted and invaluable, each typically included one pair of test inputs and could not sufficiently exercise the SUT's behaviors. The new source inputs generated by test generation techniques and the corresponding follow-up inputs enabled by MR-Adopt may reach program states not covered by the hard-coded inputs.

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

Answer to RQ4: Test cases constructed from generalized MRs could achieve 13.52% and 9.42% increases in the line coverage and mutation score, respectively, demonstrating generalized MRs's practical usefulness in enhancing test adequacy.

4.9 Threads to Validity

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

Representativeness of Experimental Subjects. A potential threat is whether our evaluation findings can generalize to different projects. To mitigate this threat, we adopted the criteria from existing studies [14, 39, 44] to select high-quality and well-maintained Java projects as representative subjects (Section 4.2) and evaluated

[12]https://pitest.org/

our method on these projects. Besides, evaluating LLMs with subjects seen during model training (known as the data leakage issue) will make the findings biased [42]. To mitigate this threat, we collected MR-encoded test cases created after the training cut-off date of the experimental LLMs, as described in Section 4.2.

Representativeness of Experimental LLMs. Our method depends on LLMs, and we also use LLMs as baselines. A potential threat is whether our evaluation findings based on the selected LLMs are representative. To mitigate this threat, we evaluated our method with LLMs from three well-known LLM families, i.e., GPT-3.5 from OpenAI, Llama3 from Meta, and Deepseek from DeepSeek. They represent the state-of-the-art code LLMs (according to the EvalPlus leaderborad) that can be deployed with the hardware capacity of our machine, as introduced in Section 4.3.

Quality of the Experimental Source Inputs. As introduced in Section 4.4, we used an LLM to prepare new source inputs to assess the generalizability of generated input transformations. Low-quality source inputs may threaten the evaluation validity. To mitigate this issue, we employed another SOTA code LLM (i.e., Qwen) which is not the experimental subject to prepare the source inputs. We then use the ground truth input transformations to filter out invalid source inputs.

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

5 RELATED WORK

5.1 Automated Identification of MRs.

Identification of proper MRs is a key step in applying MT to specific SUTs. To efficiently identify MRs, many automated approaches have been proposed. Earlier approaches identify MRs based on a set of predefined patterns [31, 52]. Zhang et al. [50] and Zhang et al. [49] proposed search-based approaches to inferring MRs. Tsigkanos et al. [38] proposed to use LLMs to identify variable relation and input transformation in scientific software. These approaches mainly synthesize MRs for specific domains. Shin et al. [33] proposed an approach to generating executable MRs from requirement specifications using LLMs, but it still requires human effort to implement supportive functions. Recently, Xu et al. [44] explored a new source to automatically derive MRs. They synthesize MRs from existing test cases where domain knowledge is embedded. This served as an effective approach to reusing many encoded MRs. Such encoded MRs are prevalent, but over 70% lack an input transformation function to support reusing them on more source inputs.

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

5.2 LLMs for Test Generation.

Researchers explored various LLM usages for test generation. Yuan et al. [48] studied the performance and limitations of ChatGPT in unit test generation. Xia et al. [43] built a fuzzer using LLMs as a generator of realistic test inputs and an engine for mutation. Tang et al. [36] compared the effectiveness of ChatGPT and Evosuite in unit test generation. Lemieux et al. [18] and Yang et al. [45] tried to promote the coverage of the tests generated by LLMs.

Different from these works, MR-Adopt does not use LLMs to generate tests directly. Instead, it generates the input transformation for the encoded MRs and reuses such MRs to enable more tests. In fact, using LLMs to generate correct and effective oracles and produce a large number of tests is found challenging [48]. In comparison, we reuse the human-written oracles in the encoded MRs, which are generally more reliable than LLM-generated oracles. Besides, MRs can be integrated with test input generation tools to produce abundant tests.

5.3 Enhancing LLMs for Code Generation.

LLMs are found powerful in code generation [19, 21], attracting numerous efforts to enhance the coding ability further. Some researchers designed more effective strategies of pre-training [12, 20, 23] and fine-tuning [6, 32]. Researchers also prompted LLMs with compilation messages to guide them to revise the generated code [16, 27, 48] or built a coding agent [51] to enhance LLM's code generation ability.

In light of prompting with analogical reasoning [47], our work guides LLMs to generate more examples, identify the intention, and finally generate an input transformation matching the intention. Also, different from the approaches that rely purely on LLMs, we enhance the generated input transformation's quality by performing data-flow analysis to exclude irrelevant code segments from LLMs' responses and ranking the generated transformation functions based on validation with the output relation.

6 CONCLUSION

This paper presents MR-Adopt, an LLM-based approach to generate input transformations for MRs with hard-coded source and follow-up inputs. MR-Adopt reuses the MRs that are encoded in the test cases to generate more tests, achieving higher test adequacy. Experimental result shows that MR-Adopt can effectively generate generalizable transformations for 72% of encoded MRs, 33.33% more then using vanilla GPT-3.5. 91.21% source inputs can be assigned valid follow-up inputs by MR-Adopt, compared with 75.99% at best baseline. Also, 10.62%+ more lines can be reached by MR-Adopt, indicating its power for more exhaustive testing. Finally, an 18.91% improvement in mutation score shows MR-Adopt's potential in bug revealing.

7 DATA AVAILABILITY

We released the implementation and all publicly available data at https://mr-adopt.github.io/.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. https://ieeexplore.ieee.org/document/6679385/

[2] Jialun Cao, Wuqi Zhang, and Shing-Chi Cheung. 2024. Concerned with Data Contamination? Assessing Countermeasures in Code Language Model. *CoRR* abs/2403.16898 (2024). https://doi.org/10.48550/ARXIV.2403.16898 arXiv:2403.16898

[3] Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code Search is All You Need? Improving Code Suggestions with Code Search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 73, 13 pages. https://doi.org/10.1145/3597503.3639085

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, and et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[5] 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

[6] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback. *CoRR* abs/2402.01391 (2024). https://doi.org/10.48550/ARXIV.2402.01391 arXiv:2402.01391

[7] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 81:1–81:13. https://doi.org/10.1145/3597503.3639219

[8] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *CoRR* abs/2401.01701 (2024). https://doi.org/10.48550/ARXIV.2401.01701 arXiv:2401.01701

[9] 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

[10] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 39:1–39:13. https://doi.org/10.1145/3597503.3608134

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024). https://doi.org/10.48550/ARXIV.2401.14196 arXiv:2401.14196

[13] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygGQyrFvH

[14] 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

[15] Maliheh Izadi, Jonathan Katzy, Tim van Dam, Marc Otten, Razvan Mihai Popescu, and Arie van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 79:1–79:13. https://doi.org/10.1145/3597503.3639138

[16] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *CoRR* abs/2306.02907 (2023). https://doi.org/10.48550/ARXIV.2306.02907 arXiv:2306.02907

[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[18] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pretrained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[19] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of Code: Reasoning with a Language Model-Augmented Code Emulator. *CoRR* abs/2312.04474 (2023). https://doi.org/10.48550/ARXIV.2312.04474 arXiv:2312.04474

[20] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, and et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[21] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, and et al. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814

[22] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Ardal, and Robert E. Wiegand. 2015. Metamorphic Model-Based Testing Applied on NASA DAT - An Experience Report. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 129–138. https://doi.org/10.1109/ICSE.2015.348

[23] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *CoRR* abs/2306.08568 (2023). https://doi.org/10.48550/ARXIV.2306.08568 arXiv:2306.08568

[24] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. 2024. KnowLog: Knowledge Enhanced Pre-trained Language Model for Log Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 32:1–32:13. https://doi.org/10.1145/3597503.3623304

[25] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 143:1–143:12. https://doi.org/10.1145/3597503.3639200

[26] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 97:1–97:13. https://doi.org/10.1145/3597503.3639187

[27] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 26106–26128. https://proceedings.mlr.press/v202/ni23b.html

[28] 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

[29] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[30] 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

[31] 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

[32] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. *CoRR* abs/2307.14936 (2023). https://doi.org/10.48550/ARXIV.2307.14936 arXiv:2307.14936

[33] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. 2024. Towards Generating Executable Metamorphic Relations Using Large Language Models. *CoRR* abs/2401.17019 (2024). https://doi.org/10.48550/ARXIV.2401.17019 arXiv:2401.17019

[34] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[35] 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

[36] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering* (2024), 1–19. https://doi.org/10.1109/TSE.2024.3382365

[37] MR-Adopt. 2024. *MR-Adopt*. Retrieved June 6, 2024 from https://mr-adopt.github.io/

[38] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software. In *Computational Science - ICCS 2023 - 23rd International Conference, Prague, Czech Republic, July 3-5, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14073)*, Jirí Mikyska, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 321–335. https://doi.org/10.1007/978-3-031-35995-8_23

[39] 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

[40] Taylor Webb, Keith J Holyoak, and Hongjing Lu. 2023. Emergent analogical reasoning in large language models. *Nature Human Behaviour* 7, 9 (2023), 1526–1541.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[42] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1282–1294. https://doi.org/10.1145/3597926.3598135

[43] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. https://doi.org/10.1145/3597503.3639121

[44] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Trans. Softw. Eng. Methodol.* (Apr 2024). https://doi.org/10.1145/3656340 Just Accepted.

[45] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *CoRR* abs/2404.04966 (2024). https://doi.org/10.48550/ARXIV.2404.04966 arXiv:2404.04966

[46] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *CoRR* abs/2404.14646 (2024). https://doi.org/10.48550/ARXIV.2404.14646 arXiv:2404.14646

[47] Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H. Chi, and Denny Zhou. 2023. Large Language Models as Analogical Reasoners. *CoRR* abs/2310.01714 (2023). https://doi.org/10.48550/ARXIV.2310.01714 arXiv:2310.01714

[48] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *CoRR* abs/2305.04207 (2023). https://doi.org/10.48550/ARXIV.2305.04207 arXiv:2305.04207

[49] 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

[50] 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

[51] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. *CoRR* abs/2401.07339 (2024). https://doi.org/10.48550/ARXIV.2401.07339 arXiv:2401.07339

[52] 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