Metamorphic Coverage

JINSHENG BA, ETH Zurich, Switzerland
YUANCHENG JIANG, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

Metamorphic testing is a widely used methodology that examines an expected relation between pairs of executions to automatically find bugs, such as correctness bugs. We found that code coverage cannot accurately measure the extent to which code is validated and mutation testing is computationally expensive for evaluating metamorphic testing methods. In this work, we propose Metamorphic Coverage (MC), a coverage metric that examines the distinct code executed by pairs of test inputs within metamorphic testing. Our intuition is that, typically, a bug can be observed if the corresponding code is executed when executing either test input but not the other one, so covering more differential code covered by pairs of test inputs might be more likely to expose bugs. While most metamorphic testing methods have been based on this general intuition, our work defines and systematically evaluates MC on five widely used metamorphic testing methods for testing database engines, compilers, and constraint solvers. The code measured by MC overlaps with the bug-fix locations of 50 of 64 bugs found by metamorphic testing methods, and MC has a stronger positive correlation with bug numbers than line coverage. MC is 4x more sensitive than line coverage in distinguishing testing methods' effectiveness, and the average value of MC is 6x smaller than line coverage while still capturing the part of the program that is being tested. MC required 359x less time than mutation testing. Based on a case study for an automated database system testing approach, we demonstrate that when used for feedback guidance, MC significantly outperforms code coverage, by finding 41% more bugs. Consequently, this work might have broad applications for assessing metamorphic testing methods and improving test-case generation.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **General and reference → Metrics**.

Additional Key Words and Phrases: Coverage metric, Metamorphic testing

ACM Reference Format:
Jinsheng Ba, Yuancheng Jiang, and Manuel Rigger. 2018. Metamorphic Coverage. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Testing identifies bugs during software development and evolution, and it typically accounts for half of the development expenses [24]. To automate testing, multiple methods have been proposed that automatically generate or mutate inputs [21, 36], on which a so-called *test oracle* is applied to determine whether the program's execution behavior is expected [6].

Metamorphic testing is a popular methodology to tackle the test oracle problem [10, 50]. It has been applied successfully in various domains, such as database systems [44, 45], compilers [32–34],

Authors' Contact Information: Jinsheng Ba, ETH Zurich, Switzerland; Yuancheng Jiang, National University of Singapore, Singapore; Manuel Rigger, National University of Singapore, Singapore.

Listing 1. A faulty absolute-difference algorithm implementation.

```
int calculate_difference(int x, int y) {
    if (x > y) {
        return x - y;
    } else {
        return y - x + 1; // 🐞 y - x;
    }
}
```

and Satisfiability Modulo Theory (SMT) solvers [42, 56, 57]. Metamorphic testing has been proposed as a methodology to test *untestable* systems, that is, systems for which it is difficult to specify the exact behavior that is expected. Rather than providing a concrete expected output for a given input, metamorphic testing uses a test input t_a to derive a new input t_b, for which a test oracle can be provided that checks whether their outputs O_a and O_b comply with a relation, which is called *Metamorphic Relation*. While metamorphic testing can be used to validate non-functional properties such as performance [5] or information leakage [40], we focus on correctness in this paper.

When designing metamorphic testing methods, it is crucial to be able to assess their effectiveness, especially for the researchers who develop such methods. According to our study on ten representative metamorphic testing methods, the most widely used metric to evaluate metamorphic testing methods is the number of found bugs. However, this metric can only be measured *a posteriori*, that is, after all the bugs found by metamorphic testing have been fixed, as identifying unique bug-inducing test cases for incorrect-output bugs is an open problem [43, 58]. This is an issue for researchers, who might want to gauge the potential of a metamorphic relation, before conducting a large-scale testing campaign, which often spans over multiple months [18]. Additionally, the quality of target programs can affect the number of unique bugs as a metric because an effective testing method cannot find many bugs in a well-tested program. Existing *a priori* metrics, which can gauge the potential of a method before conducting a large-scale testing campaign, have been sparsely adopted. Code coverage is not often used, presumably because it more precisely captures how effective a test input generator is. Mutation testing, despite advances in improving its efficiency [41], is still often prohibitively expensive to use in practice.

In this paper, we propose *Metamorphic Coverage* (*MC*), a simple and practical metric for evaluating the quality of metamorphic testing methods. We believe that a test input pair t_a and t_b is typically most effective in finding bugs when the inputs exercise different code paths, since a faulty location might be covered by t_a or t_b, but not the other, resulting in potential violations of the metamorphic relation. Therefore, our idea is to examine the difference in the code exercised by t_a and t_b to measure the quality of metamorphic relations and metamorphic testing methods. *MC* is a coverage metric based on code coverage, which can be any coverage criteria, such as line coverage. Unlike the number of bugs, which is *a posteriori* metric, *MC* is *a priori* metric that can be measured before conducting a bug-finding campaign. Compared to other coverage metrics, *MC* can more accurately measure the effectiveness of metamorphic relations. Compared to mutation testing, *MC* is a lightweight method as it does not require additional execution effort.]

Listing 1 shows a motivating example. The function calculates the absolute difference between two numbers. We consider two metamorphic relations. R_1: if $t_a = (x, y)$ and $t_b = (y, x)$, $O_a = O_b$, indicating swapping both input numbers should output the same result. R_2: for an integer c, if $t_a = (x, y)$ and $t_b = (x + c, y + c)$, $O_a = O_b$, indicating a constant added to both input numbers should output the same result. For given inputs x and y, if $x \neq y$, t_a and t_b of R_1 cover the `if` and `else` branches respectively, while t_a and t_b of R_2 cover the same `if` or `else` branch. Given $t_{a1} = (2, 3)$,

$t_{a2} = (6, 2)$ and $c = 1$, R_1 derives $(3, 2)$ and $(2, 6)$, and R_2 derives $(3, 4)$ and $(7, 3)$. Both R_1 and R_2 cover all code lines: $Cov(t_a) \cup Cov(t_b) = \{2 - 7\}$. However, only R_1 can identify the bug in line 5, which is caused by redundant `+1`. Thus, line coverage cannot distinguish the bug-finding effectiveness of both relations. Suppose *MC* is based on line coverage, the code covered by *MC* is $MC(t_a, t_b) = Cov(t_a) \triangle Cov(t_b) = \{3, 5\}$ for R_1 and $\varnothing$ for R_2. This suggests that R_1 has a higher *MC* and better bug-finding effectiveness than R_2, corresponding to our intuition that a relation is more effective in finding bugs when the inputs execute different code paths.

The intuition that the effectiveness of a metamorphic testing method depends on whether the inputs on which the metamorphic relations are defined cover disjoint portions of the code is not new. We believe most metamorphic testing methods are designed based on this intuition [32, 37, 44]. However, this intuition was only informally observed, claimed, or studied on small, artificial metamorphic relations and programs [3, 8, 11]. We systematically evaluated it on widely used metamorphic testing approaches. We also propose to utilize this intuition in other scenarios, such as guidance-based fuzzing.

We evaluated *MC* on five metamorphic testing methods. The results show that *MC* is strongly correlated to the bugs found by metamorphic testing methods, since the code covered by *MC* overlaps with the fixes of 50 of 64 bugs found by the five metamorphic testing methods. *MC* is 4× more sensitive than line coverage to distinguish method effectiveness and has the same magnitude of time consumption as line coverage. We used *MC* as guidance for generating test cases, and it could help the metamorphic testing methods *NoREC* [44] and *TLP* [45] in finding 41% more bugs than code coverage. This finding has potentially broad implications, enabling efficient feedback-driven test-case generators for metamorphic testing. The artifact is available at https://figshare.com/s/7d3a5e04c69b06204b7e.

Overall, we make the following contributions:

- We propose *MC*, a novel method to evaluate metamorphic testing methods by measuring the differential code coverage of a pair of test inputs.
- We implemented and evaluated *MC* in a comprehensive study on five metamorphic testing methods. The fixes of 50 of 64 bugs found by these methods overlap with the code measured by *MC*.

2 Background

Code coverage. Code coverage is the percentage of the source code of a program executed by a particular testing method or test suite. An assumption is that a testing method that covers more code can find more bugs, so code coverage is typically used to assess a testing method's adequacy. Multiple code coverage criteria have been proposed [20], and in this paper, we consider the following common coverage criteria:

- *Line coverage*, which measures the percentage of source code lines that have been executed;
- *Statement coverage*, which measures the percentage of instructions that have been executed;
- *Branch coverage*, which measures the percentage of control-flow branches (*e.g.* in `if` or `switch-case` statements) that have been executed.
- *Function coverage*, which measures the percentage of functions that have been executed.

To evaluate a metamorphic relation, typically, the cumulative coverage of executing all pairs of t_a and t_b is measured [44, 57]. In Listing 1, line coverage is 100% (6/6) as all lines have been executed: $\{2 - 7\}$. Statement coverage is the same as line coverage, because each line only has one statement. Branch coverage is 100% (2/2), because both branches in lines 3 and 5 have been executed. The function `calculate_difference()` is executed, so the function coverage is 100% (1/1). Code coverage cannot distinguish both metamorphic relations as both have the same coverage.

Mutation testing. Mutation testing (also known as mutation analysis) is another methodology for assessing the adequacy of a test suite by injecting mutations into programs [16, 29], and measuring whether the test suite or testing method can identify them as bugs. Mutation testing requires a set of predefined mutation operators $\{m|m \in M\}$, such as mutating an arithmetic operator $-$ to $+$, or removing a function call. We apply M to a program P, and obtain a set of program mutations $\{m(P)|m \in M\}$, each of which slightly differs from P and aims to simulate a bug. Given a test suite X, if any test $x \in X$ fails when running a $m(P)$, then $m(P)$ is said to be *killed* by X, which we denote as $kills(m(P), x)$. Otherwise, $m(P)$ is said to *survive* for X. We expect that X can kill more of $\{m(P)|m \in M\}$, so the adequacy of the test suite X is defined by the mutation score, which is computed as the fraction of program mutations killed: $\frac{|\{m(P)|m \in M \& \exists x \in X : kills(m(P),x)\}|}{|\{m(P)|m \in M\}|}$. To evaluate the quality of metamorphic relations, mutation testing can be used to examine how many simulated bugs can be identified as bugs by the metamorphic relations. In Listing 1, suppose the bug in line 5 is a mutation, executing any input violates R_1. This mutation is killed and the mutation score is 100% (1/1). The mutation score is typically deemed a good indication of the fault detection ability of a test suite [2, 20]. However, injecting mutations to simulate bugs is time-intensive [23], because the program needs to be recompiled and executed for each mutant.

3 Metamorphic Evaluation Study

As a motivating study, we investigated what metrics were used to evaluate popular metamorphic testing methods.

Studied metamorphic testing methods. As shown in Table 1, we chose ten representative metamorphic testing methods. They were published in well-known academic conferences of programming languages and software engineering during the past 10 years (2014–2024). *Referentially Transparent Inputs* (*RTI*) [25] and *Metamorphic Object Detection* (*MetaOD*) [53] test AI systems. *Equivalence Modulo Inputs* (*EMI*) [32] and *HirGen* [37] test the compilation correctness of compilers. *Non-optimizing Reference Engine Construction* (*NoREC*) [44] and *Ternary Logic Partitioning* (*TLP*) [45] test the query processing functionality of Database Management Systems (DBMSs). *YinYang* [57] and *Skeletal Approximation Enumeration* (*SAE*) [59] test Satisfiability Modulo Theory (SMT) solvers, which are used to determine whether there exists an assignment to variables that satisfy a given formula. *Excessive Data Exposure Fuzzing* (*EDEFuzz*) [39] and *Metamorphic Relation Output Patterns* (*MROP*) [49] test whether Web applications return complete and correct content. To study how these metamorphic testing methods were evaluated, we carefully examined the papers describing the ten metamorphic testing methods.

Results. The column *Metrics* in Table 1 shows the four identified evaluation metrics used for evaluating the ten studied metamorphic testing methods. *Bugs* refers to the number of found bugs in the real world. *Time* represents the execution throughput. Overall, *Bugs* is the most prevalent and important evaluation metric as it was used to evaluate all metamorphic testing methods. Three methods *EMI*, *HirGen*, and *NoREC* were evaluated by *Bugs* only. *Code Coverage* and *Mutation Score* are not commonly used, and we believe that the reason for this is the accuracy of code coverage and time cost for mutation testing. For example, concerning code coverage, in *NoREC* [44], the authors claimed that *"code coverage is not particularly useful for fuzzing DBMS, since high coverage for the core components (e.g., the query optimizer) can be achieved quickly."* *TLP* and *YinYang* measure line coverage, while *SAE* measures line, function, and branch coverage. Only *MROP* reported a mutation score. We suspect that other approaches did not adopt mutation testing due to the high execution time needed to compute it. It shows that only the number of bugs, as a *a posteriori* metric, is widely used, and no *a priori* metric is widely used—we understand an a priori metric as one that eschews an extensive testing campaign that requires developers to fix reported issues to measure

Table 1. Metrics for evaluating metamorphic testing methods. *Cov...* is short for *Code Coverage*, which includes line, function, and branch coverage. *Mutat...* is short for *Mutation Score*.

			Metrics			
Method	**Target**	**Publication**	**Bugs**	**Cov...**	**Mutat...**	**Time**
RTI [25]	AI	ICSE'20	✓			✓
MetaOD [53]	AI	ASE'20	✓			✓
EMI [32]	Compiler	PLDI'14	✓			
HirGen [37]	Compiler	ISSTA'23	✓			
NoREC [44]	DBMS	OOPSLA'20	✓			
TLP [45]	DBMS	FSE'20	✓	✓		
YinYang [57]	SMT	PLDI'20	✓	✓		
SAE [59]	SMT	FSE'21	✓	✓		✓
EDEFuzz [39]	Web	ICSE'24	✓			✓
MROP [49]	Web	ICSE'18	✓		✓	✓

the metric's effectiveness. Additionally, we found that half of the studied methods were evaluated by throughput, which shows that testing efficiency is also an important factor for metamorphic testing methods.

No widely applicable *a priori* metric is used for evaluating metamorphic testing methods.

4 Approach

We propose *Metamorphic Coverage* (*MC*), a novel metric for evaluating the quality of metamorphic testing methods by measuring the differential code executed by pairs of test inputs. For a pair of test inputs $t = (t_a, t_b)$, our intuition is that a bug can be observed if it affects the execution of either t_a or t_b but not the other, implying that ts that cover more differential code are more likely to find bugs.

We define *Metamorphic Coverage (MC)* as follows: For an ordered pair of test inputs $t = (t_a, t_b)$, the code covered by metamorphic coverage is $MC(t) = Cov(t_a) \bigtriangleup Cov(t_b) = (Cov(t_a) \cup Cov(t_b)) - (Cov(t_a) \cap Cov(t_b))$, in which $\bigtriangleup$ represents the differential coverage and $Cov(t_a)$ and $Cov(t_b)$ represent the code covered by code coverage—any concrete code coverage metric, such as line and branch coverage can be used—of executing t_a and t_b in the target system. Subsequently, for conciseness, we refer to $Cov(t_a) \bigtriangleup Cov(t_b)$ as *differential coverage*. Suppose $T = \{t_1, t_2, ..., t_k\}$, the code covered by metamorphic coverage is $MC(T) = \bigcup_{i=1}^{k}(MC(t_i))$. Figure 1 illustrates how to compute *MC*, and we explain the concrete steps to measure *MC* for each t and to combine them as follows.

Step 1: collecting coverage. Given a pair of test inputs $t = (t_a, t_b)$, we first execute the target program passing them as inputs, and measure their executed code for code coverage $Cov(t_a)$ and $Cov(t_b)$ separately. $Cov()$ can be any metric, including line $Cov_{line}()$, branch $Cov_{branch}()$, and function coverage $Cov_{function}()$. *MC* is not restricted to any specific coverage metrics. In Figure 1, we show the example by $Cov_{line}()$ because it is straightforward to understand. Unless specified, $Cov()$ is short for $Cov_{line}()$. In Figure 1, $Cov(t_a) = \{2, 4, 5, 6, 7\}$, and $Cov(t_b) = \{2, 3, 4, 6, 7\}$.

Step 2: deriving MC for t. We calculate the code covered by *MC* by measuring the differential coverage $Cov(t_a) \bigtriangleup Cov(t_b)$. In Figure 1, $MC(t) = \{3, 5\}$.

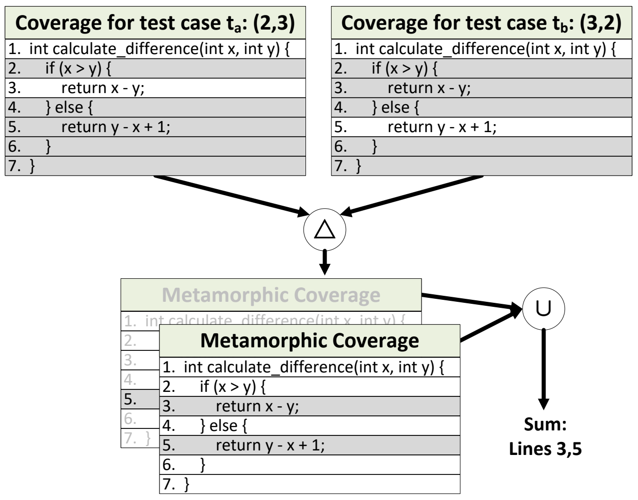
Coverage for test case t_a: (2,3)
1. int calculate_difference(int x, int y) {
2. if (x > y) {
3. return x - y;
4. } else {
5. return y - x + 1;
6. }
7. }
Coverage for test case t_b: (3,2)
△
Metamorphic Coverage
∪
Sum: Lines 3,5

Fig. 1. Overview of *MC*. The gray color above refers to the code covered by executing the program passing t_a and t_b, and the gray color below refers to the code covered in differential coverage.

Step 3: deriving MC for T. Step 2 measures the tested program parts, and we combine all $MC(t)$ by unioning them to derive $MC(T)$ for measuring the tested program parts. In Figure 1, T includes two pairs of test inputs. The *MC* for the other pair of test inputs is $\{5\}$, so $MC(T) = \{3, 5\}$.

Advantages. *MC* is effective and lightweight in evaluating the quality of metamorphic testing methods. *MC* considers the code that is validated by metamorphic relations, instead of all executed code. Therefore, *MC* is more sensitive to distinguishing the quality of different metamorphic testing methods than code coverage. Computing *MC* does not require recompiling the program, which is necessary for mutation testing, so *MC* is more lightweight than mutation testing. *MC* is slightly more expensive than code coverage as computation effort is required to derive differential code coverage.

Limitations. *MC* is not an absolute indicator of bug-finding effectiveness. First, bugs can be found even if *MC* is zero. Suppose we have a one-line C program `int output = input + 1000;`. A possible metamorphic relation is that if an input t_a is bigger than another input t_b, t_a's output must be bigger than t_b's output. A bug occurs if t_a is too big so that $t_a + 1000$ overflows. However, in this case, *MC* is zero, as both cases execute the same code. However, in Section 5, we will show that this situation is not prevalent in practice. Second, an *MC* score of 100% does not mean the target program is tested thoroughly. If t_a covers all code, while t_b does not trigger any program logic, *MC* is 100%. Typically, we expect executing both inputs to trigger different logic, so that we can compare them to find bugs. If either input does not trigger any logic, no logic is evaluated and the metamorphic relation is too weak.

Implementation. We implemented *MC* for the C/C++ programming language. For step 1, we measured code coverage by *gcov*,[1] which is the most widely used source code coverage analysis tool for C/C++. For step 2, we derive the coverage difference by implementing a plugin in *gcovr*,[2] which is a popular utility for managing *gcov*. *gcovr* supports multiple formats to store, analyze, and visualize code coverage. We used the JSON format to store the code coverage collected from step 1, because it is a structured format that facilitates machine processing. In step 2, our plugin

[1] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[2] https://gcovr.com

derives a new coverage file in the same format of JSON, so that we can leverage *gcovr* to visualize and analyze *MC* without additional implementation effort. The plugin was implemented in around 100 lines of Python code, suggesting that its low implementation effort might make the approach widely applicable.

5 Evaluation

To evaluate the effectiveness and efficiency of *MC* for evaluating metamorphic testing methods, we seek to answer the following questions:

Q.1 Effectiveness. Can *MC* evaluate the bug-finding capability of metamorphic testing methods?
Q.2 Sensitivity. To what extent can *MC* distinguish the bug-finding capability?
Q.3 Efficiency. What is the performance overhead of *MC*?
Q.4 *MC*-guided Fuzzing. Can *MC*-based feedback guide test case generation?
Q.5 Configuration Sensitivity Analysis. How does *MC* perform under different configurations?

Evaluated metamorphic testing methods. Within the studied methods in Table 1, we chose five metamorphic testing methods whose source code and bug lists are publicly available for evaluation and analysis of the effectiveness of *MC* based on found bugs. The chosen methods test three important categories of programs: DBMSs, compilers, and SMT solvers.

For DBMSs, we chose *NoREC* and *TLP*. *TLP* includes five metamorphic relations for testing the five SQL features: aggregate functions, the queries containing the clauses `DISTINCT`, `WHERE`, `GROUP BY`, and `HAVING`. We named them TLP_a, TLP_d, TLP_w, TLP_g, and TLP_h.

For compilers, we chose *HirGen* [37]. We identified two metamorphic relations in *HirGen*: $HirGen_{opt}$ and $HirGen_{mut}$, both of which were illustrated in Section 3.3.2 of its paper [37]. Although *EMI* found more bugs than *HirGen*, we did not consider *EMI*, because its source code is not available and its bug reports include minimized bug-inducing test inputs, which we cannot use to evaluate *MC*.

For SMT solvers, we chose *YinYang* [57] and *SAE* [59]. We identified two metamorphic relations in *YinYang*: $YinYang_{sat}$ and $YinYang_{unsat}$, both of which realize *YinYang* for testing satisfied and unsatisfied formulas.

Tested programs for evaluated metamorphic testing methods. Each metamorphic testing method may be applied to more than one target program, for which we chose commonly tested programs. For DBMSs, we chose SQLite and DuckDB, which were tested by *NoREC* and *TLP*. SQLite is the most widely deployed DBMS,[3] and DuckDB is a successor of SQLite. For compilers, we chose TVM [9], which was tested by *HirGen*. For SMT solvers, we chose Z3 [15] and CVC4 [7], which were tested by *YinYang* and *SAE*. TVM is a popular deep-learning compiler that optimizes the performance of AI modules. Z3 and CVC4 are the two most popular SMT solvers that regularly achieve high ranks in SMT competitions.[4] The chosen programs are written in C/C++, for which we can leverage mature code coverage tools, such as *gcov*.

Versions of tested programs. For Q1, we used SQLite (version 3.29.0), DuckDB (commit: a09d2f4 and bc9f086), TVM (commit: 124813f), Z3 (version 4.8.13), and CVC4 (commit: 16c2fe5), which correspond to the major versions in which the bugs from the bug lists were found. For Q2, Q3, and Q5, we used the latest versions supported by these metamorphic testing methods: SQLite (commit c66c77), DuckDB (version 0.5.1), TVM (commit 124813f), Z3 (version 4.13.0), and CVC4 (commit 40eac7f). For Q4, we used the last versions of programs in which the metamorphic testing methods can find bugs: SQLite (commit: 3a461f) and DuckDB (version 0.5.1).

[3]https://www.sqlite.org/mostdeployed.html
[4]https://smt-comp.github.io

Seeds for evaluated metamorphic testing methods. The evaluated metamorphic testing methods require seed inputs from which new test cases are generated. We used the default seeds used in the evaluation of each method. Specifically, *YinYang* and *SAE* adopt the conventional SMT-LIB 2 benchmark as seeds, while other methods implement custom generators to construct suitable seeds, since they require specific input structures. For example, *NoREC* mandates the presence of a `WHERE` clause, but does not support queries with `GROUP BY`, which makes conventional test suites such as TPC-DS unsuitable for its evaluation.

Baselines. We compared *MC* against line coverage and mutation score. We measured line coverage by *gcov*, and mutation score by *Mull* [17]. *Mull* is a state-of-the-art mutation testing framework that enhances performance by injecting faults into programs during compilation and enabling them respectively through environment variables. We used the 18 default mutators in *Mull*, including arithmetic and comparison mutations. *Mull* identified 19,360, 14,363, 10,396, 6,628, and 21,068 mutations for SQLite, DuckDB, TVM, Z3, and CVC4, respectively. By comparing with them, we gain insights into the benefits of *MC* against code coverage and mutation score metrics for evaluating metamorphic testing methods.

Experimental infrastructure. We conducted all experiments on an AMD EPYC 7763 processor that has 64 physical and 128 logical cores clocked at 2.45GHz. Our test machine uses Ubuntu 22.04 with 512 GB of RAM, and a maximum utilization of 60 cores. We repeated experiments 10 times for statistically significant results.

Q.1 Effectiveness

We investigated whether the bug fixes overlap with the differential code, as examined by *MC*, and whether *MC* is correlated with the number of bugs. Overlap indicates that *MC* covers at least one line of the bug fix. A significant overlap and strong positive correlation would support the applicability of *MC* in evaluating the bug-finding effectiveness of metamorphic testing methods. We evaluated both based on the historical bugs found by the evaluated metamorphic testing methods.

Preprocessing test cases. We observed that the bug reports of *NoREC*, *TLP*, and *YinYang* include minimized bug-inducing test inputs, which were reduced to a single bug-inducing input, lacking the second input as well as metamorphic relation, preventing us from directly evaluating *MC*. Listing 2 shows an example bug found by *NoREC*. Lines 1–4 show the minimized test input, in which only one query exists, while *NoREC* requires a pair of queries to validate the metamorphic relation. To address this in a best-effort manner and apply *MC*, we manually converted the test cases to trigger the bug using the proposed metamorphic relations. For example, we converted the query in line 4 into a pair of queries in lines 6–7 according to the metamorphic relation of *NoREC*—the first query is a subquery of the second query and both queries should return the same result. Therefore, t_a includes lines 1–4 and 6, and t_b includes lines 1–4 and 7. Such conversion steps were not always possible, due to information lost during minimization. For example, for *NoREC* and *TLP* some test cases lacked `WHERE` clauses needed for conversion. We successfully converted 27 bugs for SQLite and 20 bugs for DuckDB. For *YinYang*, we requested 14 original bug-inducing test inputs that include metamorphic relations from the authors. For *HirGen*, the metamorphic relations were explicit in all bug-inducing test inputs, so we directly used them. For *SAE*, most bugs were crash bugs, and we did not find any bug-inducing test inputs for the metamorphic relations. Last, we processed the test inputs only when the buggy behaviors could be reproduced and their corresponding bug fixes could be found in bug reports or were provided by authors.

Bug fix overlap. Table 2 shows the number of bugs and the relations of their fixes to *MC*. Within the total of 65 bug fixes, 50 bug fixes overlap with *MC*. 76.9% of bug fixes are directly located in the code of *MC*, showing a strong correlation between both. These bugs, whose fixes do not overlap with differential code, occur because buggy functions are called in different code locations.

Listing 2. Deriving a minimized test input into a pair of test inputs that comply to the metamorphic relation.

```
CREATE TABLE t0(c0);
INSERT INTO t0(c0) VALUES (NULL);
CREATE INDEX i0 ON t0(1) WHERE c0 NOT NULL;
SELECT * FROM t0 WHERE (t0.c0 IS FALSE) IS FALSE;
------------------------⇓------------------------
SELECT COUNT(*) FROM t0 WHERE (t0.c0 IS FALSE) IS FALSE;
SELECT SUM(count) FROM (SELECT ((t0.c0 IS FALSE) IS FALSE) as count FROM t0) as
     asdf;
```

Table 2. Previously found bugs and their relations to *MC*.

Program	All	Overlapping	Non-overlapping
SQLite	27	18	9
DuckDB	20	15	5
TVM	2	2	0
Z3	14	14	0
CVC4	1	1	0
Sum:	64	50	14

Listing 3. SQLite bug a6408d42, whose fix 45ff2b1f is overlapped with *MC*.

```
1  CREATE TABLE t0(c0);
2  INSERT INTO t0(c0) VALUES (NULL);
3  CREATE INDEX i0 ON t0(1) WHERE c0 NOT NULL;
4  SELECT COUNT(*) FROM t0 WHERE (t0.c0 IS FALSE) IS FALSE; -- {0}
5  SELECT SUM(count) FROM (SELECT (t0.c0 IS FALSE) IS FALSE as count FROM t0) as
       asdf; -- {1}
6
7  --- src/expr.c
8  +++ src/expr.c
9  @@ -5034,11 +5034,11 @@
10  switch( p->op ){
11 ...
12   case TK_TRUTH: {
13    if( seenNot ) return 0;
14    if( p->op2!=TK_IS ) return 0;
15 -  return exprImpliesNotNull(pParse, p->pLeft, pNN, iTab, seenNot);
16 +  return exprImpliesNotNull(pParse, p->pLeft, pNN, iTab, 1);
17   }
18 ...
```

An example of the relation overlapping. Listing 3 shows a bug-inducing test input for a bug found by *NoREC*. The queries in lines 4 and 5 return inconsistent results, which violate *NoREC*. Lines 15 and 16 show the code fix. The gray color represents the code covered by *MC*. The bug's root cause was an incorrect assumption that x is always not null for the expression `(x IS FALSE)IS FALSE`. This assumption is specified by the last parameter of the function `exprImpliesNotNull` and is executed for the first query in line 15. The second query evaluates the expression differently and does not

Listing 4. SQLite bug 6ef984af, whose code fix 5c118617 is not overlapped with *MC*.

```
1  CREATE TABLE t0(c0 TEXT PRIMARY KEY);
2  INSERT INTO t0(c0) VALUES ('');
3  SELECT COUNT(*) FROM t0 WHERE (t0.c0, TRUE) > (CAST('' AS REAL), FALSE); -- {0}
4  SELECT SUM(COUNT) FROM (SELECT ((t0.c0, TRUE) > (CAST('' AS REAL), FALSE)) IS
       TRUE as count FROM t0) as asdf; -- {1}
5
6  --- src/expr.c
7  +++ src/expr.c
8  @@ -68,10 +68,13 @@
9  char sqlite3ExprAffinity(Expr *pExpr){
10 ...
11 +   if( op==TK_VECTOR ){
12 +     return sqlite3ExprAffinity(pExpr->x.pList->a[0].pExpr);
13 +   }
14     return pExpr->affExpr;
15 }
16 ...
17 aff = sqlite3ExprAffinity(pExpr->pLeft);
18 ...
19 pCol->affinity = sqlite3ExprAffinity(p);
```

Table 3. Pearson Correlation Coefficient of line coverage and *MC* to bug numbers.

Program	**Line Coverage**	***MC***	**Improvement**
SQLite	0.71	0.94	+0.23
DuckDB	0.86	0.94	+0.08
Z3	0.71	0.78	+0.07
		Avg:	+0.13

execute line 15. The code of *MC* covers the bug fix in line 15, which is only executed for the first query.

An example of the relation non-overlapping. Listing 4 shows another bug-inducing test input for the bug 6ef984af found by *NoREC*. Similarly, *NoREC* finds this bug, because the queries in lines 3 and 4 return inconsistent results. This bug's root cause was a missed case in the function `sqlite3ExprAffinity` as fixed by lines 11–13. Although this function is executed for both queries, it is called in different code locations. In line 17, the function call is executed for both queries, and does not trigger the is executed only for the second query, and triggers the special case in line 11, so that the bug is observed.

Bug correlation. Based on the bugs listed in Table 2, we randomly sampled varying numbers of bugs to form multiple sets. For each set, we measured line coverage and *MC*, repeating this process 50 times. Figure 2 illustrates the relationship between the number of bugs in each set and the corresponding line coverage and *MC* scores. Visually, *MC* increases more steeply than line coverage as the number of bugs grows. To quantify this observation, we computed the *Pearson Correlation Coefficient* (*PCC*) [14] between each metric (line coverage and *MC*) and the number of bugs, as reported in Table 3. *PCC* measures the linear correlation between two variables, ranging from −1 to 1, with higher values indicating stronger positive correlation. Overall, *MC* shows a stronger positive correlation with the number of bugs than line coverage as the *PCC* of *MC* is

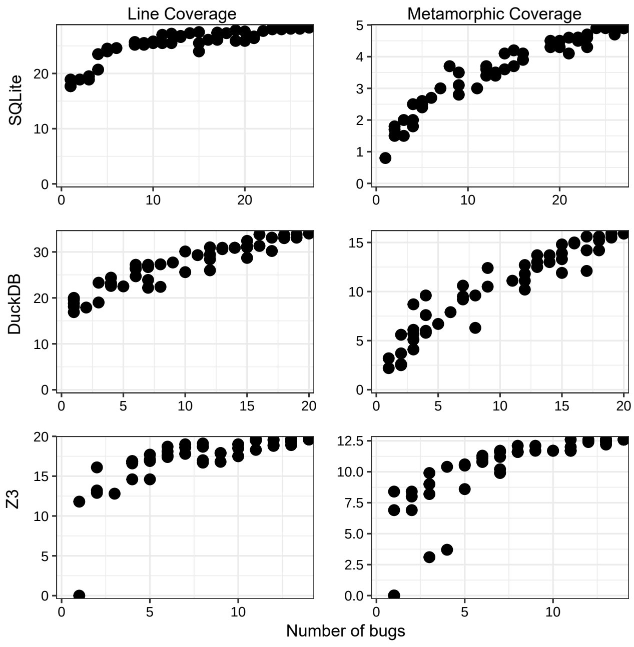

Fig. 2. Line coverage and *MC* against the number of bugs.

above 0.9 for both SQLite and DuckDB. Compared with line coverage, *MC* improves the *PCC* by an average of 0.13. We excluded TVM and CVC4 from the analysis due to an insufficient number of bugs. The mutation score was not included in Figure 2, as it consistently approaches 100%—each test case typically triggers buggy behavior, making the metric uninformative in this context.

MC is strongly correlated to the bugs found by metamorphic testing methods as *MC* overlaps with the fixes of 50 of 64 bugs found by metamorphic testing methods, and its *PCC* is above 0.9 for SQLite and DuckDB.

Q.2 Sensitivity

We evaluated the sensitivity and metric value range of *MC* for evaluating metamorphic testing methods and compared that with line coverage and mutation score. *Sensitivity* refers to the capability of differentiating test inputs and was proposed to evaluate various coverage metrics in fuzzing techniques [52]. We used the sensitivity of the various metrics to assess this aspect. Metric value range refers to the possible value range of a metric. In Q1, we showed that differential code is strongly related to bug fixes, which are in the tested program logic. Considering that line coverage subsumes *MC*, if *MC* is smaller than line coverage, *MC* can represent the bug-finding effectiveness in finer granularity.

Methodology. We measured *MC*, line coverage, and mutation scores for the test inputs generated by the evaluated metamorphic testing methods. First, we ran each metamorphic testing method to randomly generate 100 pairs of test inputs as a test suite. 100 is a widely accepted setting for test suite size and has been used for previous work [2]. 100 is also a reasonable number, as each coverage data produced by *gcov* requires one minute on average on our machine. We repeated this generation 10 times to generate 10 test suites and measured the average code coverage, mutation score, and

Table 4. The average number of line coverage, mutation score, and *MC* of metamorphic testing methods across 10 test suites.

Program	Method	Line...	Mutation...	*MC*
SQLite	TLP_w	21.84%	1.91%	1.96%
	TLP_g	22.49%	1.83%	1.87%
	TLP_h	22.61%	1.70%	1.42%
	TLP_d	22.14%	1.70%	2.14%
	TLP_a	22.79%	2.47%	2.56%
	NoREC	22.68%	1.59%	2.94%
DuckDB	TLP_w	20.45%	2.33%	3.37%
	TLP_g	21.47%	2.55%	3.82%
	TLP_h	21.17%	2.39%	3.45%
	TLP_d	21.29%	2.13%	3.52%
	TLP_a	20.99%	2.64%	4.31%
	NoREC	20.35%	1.68%	5.97%
TVM	$HirGen_{mut}$	15.30%	4.36%	3.27%
	$HirGen_{opt}$	17.53%	N/A	5.08%
Z3	$YinYang_{sat}$	14.89%	0.01%	2.84%
	$YinYang_{unsat}$	16.49%	0.01%	4.13%
	SAE	14.80%	N/A	3.75%
CVC4	$YinYang_{sat}$	22.89%	0.01%	2.85%
	$YinYang_{unsat}$	25.21%	0.01%	1.86%
	SAE	19.67%	N/A	2.17%

MC. To measure line coverage and *MC*, we first measured the line coverage for each test input of a test suite, and then calculated the cumulative coverage as the line coverage and the differential coverage as *MC* for the test suite. For mutation score, although *Mull* already implements various optimizations by injecting all mutations during compilation without the need for recompilation, the time overhead is still significant due to the number of mutants. To measure the mutation scores in a feasible time budget, we ran *Mull* in 50 parallel threads, which do not depend on each other. We deemed a mutation to be killed only when the results violate the metamorphic relation. For each testing method, we re-implemented the metamorphic relation within *Mull*, so that it can identify whether the execution results violate the metamorphic relation. Most testing methods directly produce pairs of inputs, while only *YinYang* fuses two inputs into one and examines their consistency, so its testing iteration involves three test inputs. To construct pairs of inputs, we merge the first two test inputs as t_a and deem the third input as t_b.

Challenges of measuring mutation scores. Measuring mutation scores for real-world metamorphic testing methods faces practical issues. $HirGen_{opt}$ and *SAE* had a high false alarm rate, incurring a 100% mutation score, which was meaningless, so we ignored their mutation scores. $HirGen_{opt}$ detects bugs by checking consistencies before and after optimizations. However, some optimization strategies are incompatible with specific models, and incur errors violating the metamorphic relation. We reported the false alarm issue to the authors of *HirGen*, and they confirmed our findings. *SAE* examines whether a pair of test inputs are satisfied or unsatisfied at the same time, while we observed that executing the second test input may return multiple results, instead of one. We raised GitHub issues to ask the authors of *SAE*, but have not received any response as of the time of writing this paper. For $YinYang_{sat}$ and $YinYang_{unsat}$ on Z3 and CVC4, the mutation score is around 0.01%. The reason is that *YinYang* validates the results only when the first test inputs return the

Table 5. Coefficient of variation of line coverage, mutation score, and *MC*.

Program	**Line Coverage**	**Mutation Score**	***MC***
SQLite	0.02	0.17	0.25
DuckDB	0.02	0.15	0.24
TVM	0.10	N/A	0.31
Z3	0.06	N/A	0.19
CVC4	0.12	N/A	0.22
Avg:	0.06	0.16	0.24

same SAT or UNSAT results. A random mutation can easily cause either input to become invalid or both inputs to return inconsistent results, and *YinYang* does not work for such mutated programs.

Sensitivity. To quantify the sensitivity of various metrics, we evaluated the *Coefficient of Variation* (CV) for Table 4. CV [1] measures the variability of data independently of the absolute numbers and is defined as the ratio of the standard deviation σ to the mean μ, $CV = \frac{\sigma}{\mu}$. If a metric has a higher value of CV, it is more sensitive than other metrics to distinguish the quality of test suites.

Table 5 shows our results. Across the five programs, the average CV of *MC* is 0.24, which is 4× higher than the CV of line coverage, which is 0.06. For each program, the CV of *MC* is bigger than mutation score, and the line coverage. For mutation scores, we evaluated the average value across SQLite and DuckDB, because mutation scores for other programs are not fully available. The results show that *MC* is significantly more sensitive to differentiating metamorphic relations than line coverage and also outperforms the mutation score.

Metric value range. Table 4 shows average line coverage, mutation score, and *MC* across 10 randomly generated test suites by metamorphic testing methods. Overall, *MC* is 6× smaller than line coverage. Considering that *MC* is a subset of line coverage and is strongly correlated to the found bugs, *MC* can quantify bug-finding capability in finer granularity. The relatively small size of *MC* suggests that a significant portion of the program remains untested, highlighting the potential for discovering new metamorphic relations to improve testing.

MC can efficiently distinguish the quality of metamorphic testing as it is 4× more sensitive than line coverage and mutation score for evaluating metamorphic testing. The average value of *MC* is 6× smaller than line coverage, while still capturing the tested program logic.

Q.3 Efficiency

While our major questions are about the evaluation effectiveness of *MC*, performance overhead is also an important consideration for evaluation metrics, as a high overhead might limit a metric's applicability. We evaluated the execution time of *MC*, line coverage, and mutation score based on the same experimental set-up and data in Q2. We derived the execution time by multiplying the actual execution time by 50× as we ran *Mull* in 50 parallel threads.

Results. Table 6 shows the average execution time for measuring line coverage, mutation score, and *MC* across 10 test suites in Q2. Overall, measuring *MC* consumed the same magnitude of time as measuring line coverage, and 359x less time than measuring mutation score. Mutation score consumes significant time due to its computation complexity. This is particularly noticeable on CVC4, which consumes the most time, because a random mutation can easily cause CVC4 to hang. We set a timeout of 10 minutes for *Mull* to solve this issue. Compared to mutation score, *MC* is more lightweight, because it consumes much less time and does not require test oracles for validating metamorphic relations. Compared to line coverage, *MC* is a more efficient method,

Table 6. Average time (dd:hh:mm:ss) consumption of measuring line coverage, mutation score, and *MC* on metamorphic testing methods across 10 test suites.

Program	**Line Coverage**	**Mutation Score**	***MC***
SQLite	00d:00h:33m:24s	02d:11h:20m:45s	00d:00h:36m:01s
DuckDB	00d:01h:16m:06s	02d:21h:51m:02s	00d:01h:22m:27s
TVM	00d:04h:06m:44s	05d:02h:46m:10s	00d:04h:17m:44s
Z3	00d:07h:26m:27s	11d:21h:09m:55s	00d:07h:46m:47s
CVC4	00d:05h:11m:10s	240d:01h:31m:40s	00d:05h:28m:45s
Avg:	00d:03h:18m:22s	52d:11h:31m:30s	00d:03h:30m:33s

because it can represent the checked code by a metamorphic testing method, but only moderately increases execution time.

MC is resource-efficient as it requires 359x less time than mutation score and has the same magnitude of time consumption as line coverage.

Q.4 *MC*-guided Fuzzing

Multiple fuzzing methods use metrics, such as branch coverage [54] and code execution count [35], as guidance to generate or mutate test inputs and have found a large number of bugs. Apart from evaluating the effectiveness of metamorphic testing methods, we evaluated whether *MC* can be used as guidance to generate more diverse test inputs increasing the likelihood of finding bugs.

Methodology. For the evaluated metamorphic testing methods, *NoREC* and *TLP* involve a test input generation process, while *HirGen*, *YinYang*, and *SAE* require user-provided test inputs as t_a for deriving t_b, so we evaluated whether *MC* can help *NoREC* and *TLP* generate diverse test inputs. *NoREC* and *TLP* are implemented in *SQLancer*, which generates test inputs by *Query Plan Guidance* (*QPG*) [4]. Given a database, *QPG* randomly generates queries and examines their query plans. If no new unique query plan has been seen for a fixed number of iterations, *QPG* mutates the database and continues to randomly generate queries on the new database. For a fair comparison, we used *QPG* as a reference and reused its test input generation workflow. Specifically, we replaced *QPG* as a feedback mechanism with code coverage and *MC*, and mutated the database if the coverage was not increased for a fixed number of iterations. We reused the instrumentation component of AFL++ [19] to collect branch coverage and derived *MC*. We refer to both methods as *Code Coverage Guidance* (*CCG*) and *Metamorphic Coverage Guidance* (*MCG*), respectively.

Results. Figure 3 shows the average number of bugs found by *CCG*, *MCG*, and *QPG*. DuckDB terminated before 24 hours due to crash bugs. For TLP_a in SQLite, all guidance techniques found more than 100 bug-inducing test cases in five minutes, so it cannot distinguish the contributions of guidance, and we excluded it. Overall, more bugs were found in DuckDB than in SQLite, so the gap between guidance is clearer in DuckDB. On average, *CCG* found 4.4 bugs, *MCG* found 6.2, and *QPG* found 6.5 bugs in 24 hours. *MCG* outperforms *CCG* on 10 of 11 targets. The result shows that *MCG* can help metamorphic testing methods generate more diverse test inputs for finding bugs. The only exception is TLP_a in DuckDB, in which most found bugs are unexpected errors. Unexpected errors refer to unhandled exceptions in programs, such as assertions, and can be found by a test input, instead of a pair of test inputs. *SQLancer* can find these unexpected errors by checking error information without *NoREC* and *TLP*. Therefore, *CCG* helps find more bugs. *MCG* outperforms *QPG* on 5 of 11 targets. This is unsurprising, as *QPG* was designed as a domain-specific feedback

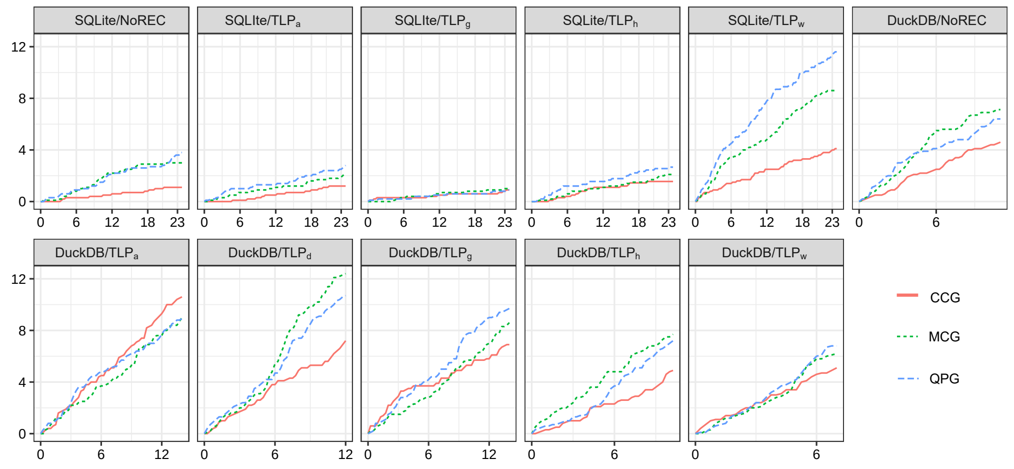

Fig. 3. Average number of bugs found by Code Coverage Guidance, Metamorphic Coverage Guidance, and Query Plan Guidance across 10 runs in 24 hours.

Table 7. Coefficient of variation of *MC* based on branch and function coverage.

	Branch Coverage		**Function Coverage**	
Program	**Branch Coverage**	***MC***	**Function Coverage**	***MC***
SQLite	0.02	0.37	0.01	0.50
DuckDB	0.02	0.32	0.02	0.35
TVM	0.10	0.26	0.12	0.15
Z3	0.19	0.20	0.07	0.22
CVC4	0.33	0.35	0.09	0.50
Avg:	0.13	0.30	0.06	0.34

metric based on the insight into how DBMSs execute test inputs, while *MC* can be applied to any metamorphic testing approach, and thus eschews additional domain-specific insights.

Using *MC* as guidance improved bug-finding efficiency by 41% on *NoREC* and *TLP* compared to code coverage, although domain-specific strategies like *QPG* often perform best.

Q.5 Configuration Sensitivity Analysis

Various code coverage metrics. We evaluated the sensitivity of *MC* using different kinds of code coverage metrics. As we explained in Section 4, *MC* is not restricted to line coverage, so we evaluated whether *MC* performs similarly under branch and function coverage. Specifically, we reused the same experimental set-up and data in Q2, and examined the CV of *MC* based on branch and function coverage.

Results. Table 7 shows the CV of *MC* based on branch and function coverage. On average, *MC*'s CV is 2.3× more than branch coverage's CV, and 5.7× more than function coverage. Similar to Table 5, *MC* is significantly more sensitive than line, branch, and function coverage to distinguish the quality of metamorphic testing methods. The result shows that *MC* consistently outperforms code coverage irrespective of at what granularity it is applied.

Table 8. Coefficient of variation of *MC* based on test suite sizes of 50 and 200.

	Size 50		Size 200	
Program	**Line Coverage**	***MC***	**Line Coverage**	***MC***
SQLite	0.02	0.36	0.02	0.37
DuckDB	0.04	0.38	0.02	0.37
TVM	0.15	0.56	0.11	0.29
Z3	0.13	0.28	0.11	0.22
CVC4	0.21	0.43	0.17	0.37
Avg:	0.11	0.40	0.09	0.32

Test suite size. For Q2, we evaluated *MC* based on test suites of 100 test inputs. To answer whether different test suite sizes affect the sensitivity of *MC*, we examined the CV of *MC* based on test suite sizes of 50 and 200, which are similar to the sizes used in the previous work [2]. Due to the resource limitation, we did not consider a bigger size.

Results. Table 8 shows the CV of *MC* based on test suite sizes 50 and 200. We observed that the results are close to the results in Table 5 that *MC* is around 4× more sensitive than line coverage. It shows that test suite sizes have no significant effect on the experiments' results. The observation that test suite size has no significant impact on metric evaluation is consistent with previous work [2, 13].

MC performs similarly under branch and function coverage, and under different test suite sizes.

6 Discussion

Path to adoption. We believe that *MC* has the potential for wide adoption. From a conceptual perspective, *MC* is easy to understand, as it calculates the differential code executed by pairs of test inputs. From an implementation perspective, *MC* is easy to implement as we implemented *MC* in around 100 lines of Python code for the C/C++ programming language. From an applicability perspective, *MC* can be implemented on top of existing widely-used coverage measurement tools in other languages, such as JaCoCo for Java, or Coverage.py for Python. From an empirical perspective, our study on widely-used metamorphic relations and systems suggests the effectiveness of *MC*.

Improving metamorphic testing methods. MC could be used to identify metamorphic relations for code that is not covered by existing metamorphic relations. From our evaluation, we found that most metamorphic testing methods focus on specific features, so their *MC* is small (*i.e.*, all evaluated metamorphic testing methods have an *MC* of less than 5%). For future research on metamorphic testing methods, we believe that systematically designing metamorphic relations to fill *MC* gaps could significantly improve the bug-finding capability.

Metamorphic relation design guided by MC. Metamorphic relation design is largely a manual and creative task, making it difficult to isolate *MC* as a quantifiable factor in an automated experiment. Rather, we show an example to show how *MC* guides the design of new metamorphic relations. In Listing 5, suppose we start with Metamorphic Relation 1 (*MR1*): $abs(x) == abs(-x)$. For example, $abs(3) == abs(-3)$ and $abs(0) == abs(-0)$. *MR1* covers the first and third branches (*i.e.*, lines 2, 3, 6, and 7) differently, so its MC covers the two branches without the second branch at lines 4 and 5. To fill this gap, we need a metamorphic relation in which one test case takes the x == 0 branch while the other takes a different branch. This means the two inputs must include zero and a non-zero value. We can propose Metamorphic Relation 2 (*MR2*): $abs(x) \geq abs(0)$. For instance, Listing 5. An example of designing new metamorphci relations guided by *MC*.

```
int abs(int x) {
    if (x < 0)
        return -x;
    else if (x == 0)
        return 3; // Bug
    else
        return x;
}
```

$abs(3) \geq abs(0)$ or $abs(-5) \geq abs(0)$. In this case, MC covers all branches, and *MR2* exposes the bug, as $abs(2) < abs(0)$.

MC on metamorphic relations with multiple inputs and outputs. In Figure 1, we defined *MC* based on metamorphic relations expressed as test pairs $t = (t_a, t_b)$, where t_a and t_b each correspond to a single input. However, some relations require multiple inputs in either t_a or t_b. For example, *YinYang*, evaluated in Section 5, defines t_a using two inputs. In such cases, we treat the union of the code coverage from all inputs in t_a as the coverage of t_a. This step would not affect the effectiveness of *MC*, as it aims to capture the differential coverage between t_a and t_b, reflecting the likelihood of exposing violations of the metamorphic relation.

Threats to Validity. The evaluation of *MC* faces potential threats to validity. A concern is internal validity, that is, the degree to which our results minimize systematic error. *MC* was compared to other metrics based on randomly generated test suites. The randomness process may limit the reproducibility of our results. To mitigate the risk, we repeated all experiments 10 times to account for potential performance fluctuations. The other concern is external validity, that is, the degree to which our results can be generalized to and across other metamorphic testing methods. We selected five representative metamorphic testing methods in three commonly tested domains. According to our study, metamorphic testing methods follow similar testing mechanisms, suggesting that our results generalize to other metamorphic testing methods.

7 Related Work

Metamorphic testing. Several studies present a comprehensive overview of metamorphic testing [12, 48, 50]. As shown in Table 1, we investigated the evaluation metrics of ten metamorphic testing methods, which have cumulatively found thousands of bugs. However, in this work, rather than studying existing metamorphic relations or proposing a new one, our core contribution is a novel coverage metric to evaluate metamorphic testing methods and a comprehensive evaluation thereof.

Evaluating metamorphic relations. Empirical studies were conducted to investigate the effectiveness of metamorphic relations. These include case studies by Chen *et al.* [11] on the shortest path program, by Asrafi *et al.* [3] on two small programs with manually crafted metamorphic relations, and by Guderlei *et al.* [38] on two small programs with manually crafted metamorphic relations. The authors of the above studies consistently observed that the metamorphic relations that cause different software execution behaviors should have high fault detection ability. However, the concept of "difference" between executions was not clearly defined. Cao *et al.* [8] studied various notions of difference and bug-finding effectiveness. However, all of the above papers studied metamorphic relations specifically designed for the study, rather than metamorphic relations that have demonstrated their bug-finding capabilities on important and well-tested systems. Unlike these works that focus on empirical analyses, in this work, we quantitatively define a metric to evaluate the quality of metamorphic testing methods and evaluate them on widely used programs and effective metamorphic relations, both of which have been tested in the real world. We also propose a *MC*-guided fuzzing to utilize *MC* to increase the likelihood of finding bugs.

Oracle coverage. Some metrics have been proposed to evaluate test oracles, mostly assertions, which typically check invariants when executing a single test input. Koster *et al.* [31] proposed state coverage, which measures the percentage of the output-relevant variables checked by an assertion. Vanoverberghe *et al.* [51] improved state coverage to measure the percentage of only program variables that are read by the assertion. Schuler *et al.* [46, 47] proposed checked coverage, which measures the percentage of statements checked by an assertion based on all statements that influence the assertion by control flow or data flow. Hossain *et al.* [26] proposed to reduce the total space of checked coverage by measuring only the statements that influence at least one value checked by the assertion. In this work, we propose *MC* to evaluate metamorphic testing, which compares discrepancies between two test inputs instead of an input.

Coverage criteria. A large body of work considers the relationship between coverage criteria and fault detection. Gligoric *et al.* [20] examined the correlations of various coverage to mutation testing and concluded that branch coverage and intra-procedural acyclic path coverage perform best to predict the mutation score, which is assumed to be a silver criterion for evaluating test suites. Gopinath *et al.* [22] further studied the problem and concluded that statement coverage performs best. Inozemtseva *et al.* [27] investigated the correlation between various coverage criteria and the mutation score for different random subsets of test suites and found a low to moderate correlation between coverage and effectiveness when the number of test inputs in the suite is controlled for. Kakarla [30] and Inozemtseva [28] demonstrated a linear relationship between mutation score and various coverage criteria for individual programs. Wei *et al.* [55] examined branch coverage as a quality measure showing that branch coverage behavior was consistent across many runs, while fault detection varied widely. Existing work mainly focuses on evaluating the correlation of various coverage criteria to mutation scores. In contrast, in this work, we propose a novel coverage criterion *MC* for evaluating metamorphic testing.

8 Conclusion

In this paper, we have proposed a novel coverage metric, *Metamorphic Coverage* (*MC*), to evaluate the quality of metamorphic testing methods. The core idea of *MC* is that a bug can be observed if the faulty code path is executed by a test input, but not the other test input. Therefore, if a metamorphic testing method covers more differential code between the execution of pairs of test inputs, it is more likely to find bugs. The results show that *MC* is strongly correlated to the bugs found by metamorphic testing methods as the code covered by *MC* overlaps with the fixes of 50 of 64 bugs found by the five metamorphic testing methods. *MC* is 4× more sensitive than line coverage in distinguishing the quality of metamorphic testing methods and is similarly lightweight in terms of time consumption as line coverage. Despite these promising results, *MC* is no panacea, similar to code coverage, as achieving a high *MC* is possible even if the metamorphic test oracle has low bug-finding effectiveness and *vice versa.* In the future, we believe that *MC* can be broadly applied to assess metamorphic testing methods and improve test-case generation by using *MC* as a metric in feedback-guided automated testing.

References

[1] Hervé Abdi. Coefficient of variation. *Encyclopedia of research design*, 1(5), 2010.

[2] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[3] Mahmuda Asrafi, Huai Liu, and Fei-Ching Kuo. On testing effectiveness of metamorphic relations: A case study. In *2011 fifth international conference on secure software integration and reliability improvement*, pages 147–156. IEEE, 2011.

[4] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *The 45th International Conference on Software Engineering (ICSE'23)*, May 2023.

[5] Jinsheng Ba and Manuel Rigger. Finding performance issues in database engines via cardinality estimation testing. In *The 46th International Conference on Software Engineering (ICSE'24)*, April 2024.

[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[7] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[8] Yuxiang Cao, Zhi Quan Zhou, and Tsong Yueh Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *2013 13th International Conference on Quality Software*, pages 153–162. IEEE, 2013.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11(2018):20, 2018.

[10] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 2020.

[11] Tsong Yueh Chen, DH Huang, TH Tse, and Zhi Quan Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583. Citeseer, 2004.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[13] Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 237–249, 2020.

[14] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4, 2009.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[17] Alex Denisov and Stanislav Pankevich. Mull it over: mutation testing based on llvm. In *2018 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pages 25–31. IEEE, 2018.

[18] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[20] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[21] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[22] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th international conference on software engineering*, pages 72–82, 2014.

[23] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. Systematic assessment of fuzzers using mutation analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4535–4552, 2023.

[24] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[25] Pinjia He, Clara Meister, and Zhendong Su. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 410–422. IEEE, 2021.

[26] Soneya Binta Hossain, Matthew B Dwyer, Sebastian Elbaum, and Anh Nguyen-Tuong. Measuring and mitigating gaps in structural testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1712–1723. IEEE, 2023.

[27] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[28] Laura Michelle McLean Inozemtseva. Predicting test suite effectiveness for java programs. Master's thesis, University of Waterloo, 2012.

[29] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[30] Sahitya Kakarla. *An analysis of parameters influencing test suite effectiveness.* PhD thesis, Texas Tech University, 2010.

[31] Kenneth Koster and David C Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 541–544, 2007.

[32] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.

[33] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 386–399. ACM, 2015.

[34] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 327–337, 2015.

[35] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[36] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[37] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 248–260, 2023.

[38] Johannes Mayer and Ralph Guderlei. An empirical study on the selection of good metamorphic relations. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 475–484. IEEE, 2006.

[39] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. Detecting excessive data exposures in web server responses with metamorphic fuzzing. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.

[40] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. Edefuzz: A web api fuzzer for excessive data exposures. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[41] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in computers*, volume 112, pages 275–378. Elsevier, 2019.

[42] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–19, 2021.

[43] Cheng Qian, Ming Zhang, Yuanping Nie, Shuaibing Lu, and Huayang Cao. A survey on bug deduplication and triage methods from multiple points of view. *Applied Sciences*, 13(15):8788, 2023.

[44] Manuel Rigger and Zhendong Su. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, 2020.

[45] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[46] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99. IEEE, 2011.

[47] David Schuler and Andreas Zeller. Checked coverage: an indicator for oracle quality. *Software testing, verification and reliability*, 23(7):531–551, 2013.

[48] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[49] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. In *Proceedings of the 40th International Conference on Software Engineering*, pages 882–882, 2018.

[50] Sergio Segura and Zhi Quan Zhou. Metamorphic testing 20 years later: a hands-on introduction. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 538–539. ACM, 2018.

[51] Dries Vanoverberghe, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. State coverage: Software validation metrics beyond code coverage. In *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current*

Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings, pages 542–553. Springer, 2012.

[52] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[53] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 1053–1065. IEEE, 2020.

[54] Website. American fuzzy lop (afl) fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2013. Accessed: 2022-06-08.

[55] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, pages 194–212, 2012.

[56] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.

[57] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation*, pages 718–730, 2020.

[58] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. Silent compiler bug de-duplication via three-dimensional analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 677–689, 2023.

[59] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for smt solver testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1141–1153, 2021.