GeMTest: A General Metamorphic Testing Framework

Simon Speth
School of Computation, Information and Technology
Technical University of Munich
Munich, Germany
simon.speth@tum.de

Alexander Pretschner
School of Computation, Information and Technology
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

***Abstract*—Metamorphic testing (MT) is an established software testing methodology suitable for testing various types of systems under test (SUTs), but identifying and implementing metamorphic relations (MRs) remains a challenge. This paper presents GeMTest, a general-purpose metamorphic testing framework that is domain-independent, enabling software testers to implement MRs in Python and execute them with pytest. In GeMTest, MRs are implemented using custom decorators to annotate Python functions, which specify the follow-up generation function, the metamorphic oracle, and the SUT. GeMTest then automatically creates and executes a pytest test suite containing multiple metamorphic test cases derived from the user-defined MRs. We evaluate GeMTest on 16 program domains, ranging from trigonometric functions to deep learning image classifiers, implementing in total over 200 MRs. To enable the adoption and encourage further extension, the open-source implementation of GeMTest is available on GitHub. Our demo video is available at https://youtu.be/Ec5SK-meu90.**

***Index Terms*—metamorphic testing, testing framework, python, pytest, tool.**

I. Introduction

Metamorphic testing (MT) [2, 3] is a widely acknowledged technique in software testing research, showing an increasing amount of publications and software domains it is applied to [3, 16]. In MT, multiple metamorphic relations (MRs) are usually created manually, against which the system under test (SUT) is then tested. In essence, a MR provides an automated pseudo-oracle, a function that indicates whether the outputs of the SUT are potentially correct or not [18]. A basic MR provides a pseudo-oracle as follows: (1) By selecting a source input, a concrete metamorphic test case (MTC) is instantiated from a MR. (2) When executing this MTC, a follow-up input is computed from the source input, such that the SUT is executed twice, once with each input. Finally, (3) the two SUT outputs are compared with a relation, also called metamorphic oracle.

Even though MT is widespread in research [16], there is a lack of solid tooling solutions to assist software testing practitioners and researchers in specifying and formulating MRs using a language in a way that MT can be automated and seamlessly integrated into their existing testing frameworks.

Although the concept of MT is relatively simple and its implementation straightforward [3], current testing frameworks like pytest [15] and JUnit [7] lack support for a structured and simple implementation of MRs. In those frameworks, MTCs cannot be automatically derived, executed, and analyzed. Multiple approaches attempt to bridge this gap for a specific domain but fail to allow test engineers to specify MRs in a general and domain-independent way [4, 6, 9, 13, 20].

```python
test_sin_metamorphic.py
import gemtest as gmt
import math

mr_1 = gmt.create_metamorphic_relation(name='periodicity', data=range(100))

@gmt.transformation(mr_1)
def plus_two_pi(source_input):
    return source_input + 2 * math.pi

@gmt.relation(mr_1)
def equals(source_output, followup_output):
    return gmt.approximately(source_output, followup_output)

@gmt.system_under_test()
def test_sin(input):
    return math.sin(input)
```

Listing 1: Python code to specify one MR in GeMTest for the $sin(x)$ function. Source inputs for creating 100 MTCs are the numbers from 0 to 99. The execution is shown in Listing 2.

```
user@host:~$ pytest test_sin_metamorphic.py
=============== test session starts ================
platform linux -- Python 3.10.12, pytest-8.3.4,
pluggy-1.5.0
rootdir: /home/user/gemtest
plugins: gemtest-1.0.0, hypothesis-6.124.1,
html-3.2.0, metadata-3.1.1, xdist-3.6.1
collected 100 items

test_sin_metamorphic.py .................. [ 18%]
.......................................... [ 60%]
........................................   [100%]

=============== 100 passed in 0.28s ================
```

Listing 2: Output when executing the metamorphic test suite definition `test_sin_metamorphic.py` with pytest.

To address this gap, we present GeMTest, a general-purpose metamorphic testing framework that enables test engineers to implement MRs in Python and automatically derive and execute MTCs as pytest test cases. Our framework is domain-independent and can be used to test any program or function that can be called within Python. Additionally, already existing test infrastructure can be used as GeMTest creates and executes a pytest test suite, enabling tool support and reusability. Tests implemented in GeMTest are designed to be short, readable, and free of unnecessary syntactic overhead, as demonstrated in Listing 1. When implementing the MR $\sin(x) = \sin(x + 2\pi)$, which verifies the periodicity property of the SUT $\sin(x)$, a

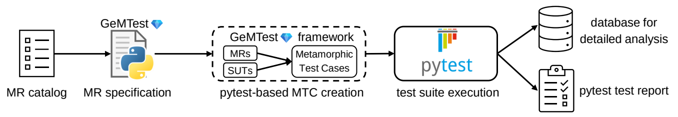


Figure 1: Workflow of a test engineer using GeMTest.

test engineer needs only 16 lines of code (LOC). The MTCs are then executed by running pytest, which also visualizes and summarizes the individual test results, as shown in Listing 2.

With GeMTest, we make the following contributions:

- We introduce a domain-specific language for specifying MRs, implemented in Python, which instantiates MTCs as pytest test cases, maximizing reusability and tool support.
- To demonstrate the domain-independent applicability of our GeMTest framework, we implement 218 MRs for 16 program domains, ranging from trigonometric functions to deep learning (DL) LiDAR object detectors [17].
- To foster further improvement and the addition of new features, we make the implementation of GeMTest available on GitHub https://github.com/tum-i4/gemtest and permanently archive GeMTest on Figshare [5].

II. Related Work

JFuzz, introduced by Zhu [20] in 2015, is a Java unit testing framework based on the principles of fuzz testing [14] and MT. Compared to GeMTest, JFuzz also enables MRs similar to the definition in [3] but does not restrict the implementation of the tests, which might lead to test cases implemented that are not necessarily MTCs. MRs in JFuzz are implemented in one single method, giving the user little structure but high flexibility. In our framework, the MR definition is split up into three methods, providing more structure without compensating flexibility.

SMRL, introduced by Mai et al. [13] in 2020, is a domain-specific framework for MT of web systems, built on top of the Java testing framework JUnit [7]. It allows users to specify MRs in Xbase, which are then translated into Java-based JUnit MTCs. Similarly, GeMTest is built on the established testing framework pytest [15], but avoids using additional languages, allowing MRs to be specified directly in Python, making GeMTest domain-independent and more accessible to testers.

GOTTEN [1, 6] is a domain-specific MT tool based on a textual domain-specific language (DSL), called mrDSL, for specifying MRs. The Java tool uses search-based techniques to generate follow-up inputs. GOTTEN requires users to create a meta-model of the domain and understand mrDSL that utilizes object constraint language. In contrast, GeMTest does not require a meta-model, making it more straightforward to apply.

Similar to GeMTest, plugins for enhancing the functionality of pytest exist, such as *pytest-inline* [10, 11] or *hypothesis* [12]. We follow this design approach to increase user adoption and ease integration into existing testing frameworks and workflows.

III. The GeMTest Framework

Our metamorphic testing (MT) framework GeMTest is a tool for software testers that automatically generates and executes MTCs based on user-implemented MRs in Python. GeMTest is designed as an extension to pytest by providing a Python-based language for specifying MRs, which correspond to the definition of MT as given by Chen et al. [3].

Following the basic example from Listing 1, we use three decorators provided by GeMTest for implementing a metamorphic test suite in pytest: (1) a function annotated with the `@transformation` decorator is registered as a metamorphic transformation taking the source input as a function input and returning the follow-up input. (2) The `@relation` decorator registers a function as the metamorphic oracle, returning a boolean value indicating whether the MTC passed or failed. (3) The `@system_under_test` decorator specifies the decorated function as the system under test (SUT). This function will be executed multiple times with different inputs to generate the source and follow-up outputs.

The MT workflow with GeMTest is shown in Figure 1: A test engineer elicits MRs for the SUT's domain and implements those MRs in Python using the decorators and methods provided by GeMTest (see Listing 1 as an example of a MR specification). By running pytest, this specification is translated into concrete pytest MTCs which are executed in the pytest framework. Finally, the tester can see the summarized results of all tests in the command line or IDE. For a detailed analysis, test results and the artifacts of each single MTC are stored in a database and can be visualized by the gemtest-webapp.

A. GeMTest Features

This section briefly summarizes the key features of GeMTest, our general-purpose metamorphic testing framework.

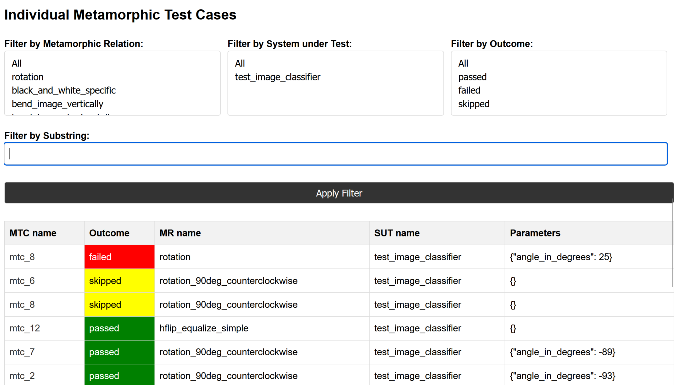
Support for General MRs: GeMTest supports all types of MRs as specified by Chen et al. [3] in the most recent and revised definition of MT. For example, in GeMTest testers can specify MRs with multiple source and follow-up inputs, past-execution dependent MRs [19] where the follow-up input depends on the source output, and MRs where the metamorphic oracle compares both inputs and outputs of the MTC.

Python-Based MR Definition: To lower the entry barrier and simplify MR implementation, GeMTest allows MRs to be written in pure Python. MRs are structured into three Python functions – transformation, relation, and system under test – each function annotated with a GeMTest decorator.

Domain Independence: Since MRs are implemented in Python, any function or class callable from within Python can be tested with GeMTest. This includes Python functions, Python packages, command line programs, and DL models.

Pytest Compatibility: GeMTest is built on top of pytest and generates pytest test cases during execution. This ensures seamlessly integration into existing workflows for test engineers already using the pytest framework for software testing.

Test Reporting and Summary: Software testers have three options for evaluating the test results: First, using the classical pytest output to get information at the level of individual MTCs (see Listing 2). Second, a *string-report* giving insights into the execution of a MTC, especially why a MTC failed. Third, a *html-report* for showing metrics, such as failure rate grouped by MR and SUT, filtering of MTCs by test verdict such as



Individual Metamorphic Test Cases

Filter by Metamorphic Relation: All / rotation / black_and_white_specific / bend_image_vertically

Filter by System under Test: All / test_image_classifier

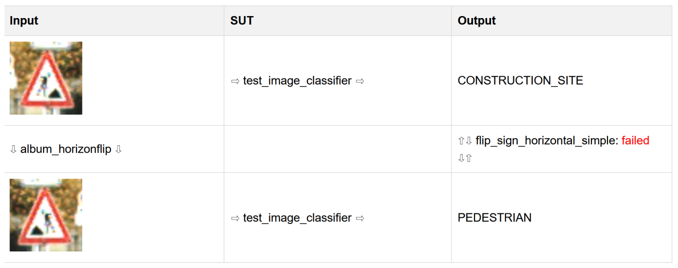
Filter by Outcome: All / passed / failed / skipped

Filter by Substring:

Apply Filter

MTC name	Outcome	MR name	SUT name	Parameters
mtc_8	failed	rotation	test_image_classifier	{"angle_in_degrees": 25}
mtc_6	skipped	rotation_90deg_counterclockwise	test_image_classifier	{}
mtc_8	skipped	rotation_90deg_counterclockwise	test_image_classifier	{}
mtc_12	passed	hflip_equalize_simple	test_image_classifier	{}
mtc_7	passed	rotation_90deg_counterclockwise	test_image_classifier	{"angle_in_degrees": -89}
mtc_2	passed	rotation_90deg_counterclockwise	test_image_classifier	{"angle_in_degrees": -93}

Figure 2: gemtest-webapp based *html-report*.



Input	SUT	Output
	⇨ test_image_classifier ⇨	CONSTRUCTION_SITE
⇩ album_horizonflip ⇩		⇧⇩ flip_sign_horizontal_simple: failed ⇩⇧
	⇨ test_image_classifier ⇨	PEDESTRIAN

Figure 3: gemtest-webapp enabled MTC detail view.

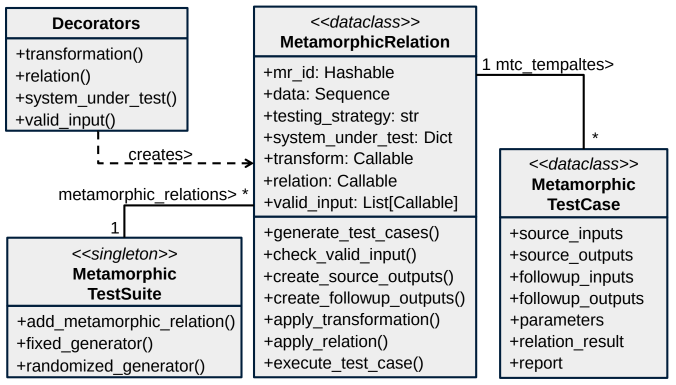


Figure 4: UML metamodel of the GeMTest architecture.

passed, failed, or skipped (see Figure 2), and detailed analysis of a single MTCs with custom visualizations (see Figure 3).

MT of DL Models: GeMTest implements several optimizations and features improving performance when testing DL models. For example, GeMTest supports input batching, which groups multiple source and follow-up inputs into a batch for faster execution and better hardware utilization. Additionally, SUTs can be specified via the command line, allowing testers to evaluate different DL models that share the same architecture.

Extensively Tested: GeMTest contains an extensive test suite implementing 489 pytest tests in 2,800 LOC. The test suite extensively tests our framework as it achieves 96% statement coverage, 95% branch coverage, and a 91% mutation score.

B. GeMTest Implementation

The GeMTest framework is implemented in Python as a pytest plugin, using a modular architecture designed for further

test_sort_metamorphic.py

```
 1 import gemtest as gmt
 2
 3 mr1 = gmt.create_metamorphic_relation(
   ↪  "add_element", data=generated_lists,
   ↪  number_of_sources=2)
 4
 5 @gmt.general_transformation(mr1)
 6 @gmt.randomized("s", gmt.RandInt(1, 10))
 7 def add_element( mtc: gmt.MetamorphicTestCase,
   ↪  s: int):
 8     a, b = mtc.source_inputs
 9     e = max(a + b) + s
10     return a + b, [e] + a + b
11
12 @gmt.general_relation(mr1)
13 def sorted(mtc: gmt.MetamorphicTestCase):
14     a, b = mtc.source_inputs
15     c, d = mtc.followup_outputs
16     e = max(a + b) + mtc.parameters['s']
17     return all(
        ↪  x == y for x, y in zip(c + [e], d))
18
19 @gmt.system_under_test()
20 def test_insertionSort(list):
21     return insertionSort(list)
22
23 @gmt.system_under_test()
24 def test_mergeSort(list):
25     return mergeSort(list)
```

Listing 3: MR specification in GeMTest for testing two sort functions. The MR has multiple source and follow-up inputs and accesses multiple MTC artifacts in the oracle.

extension. An overview of the core software architecture is depicted in Figure 4. It consists of **`Decorators`** to register user-defined Python functions for a **`MetamorphicRelation`**, specifically transformation, relation, and system under test. A **`MetamorphicRelation`** object is considered to be a blueprint for a concrete **`MetamorphicTestCase`** object such that a **`MetamorphicRelation`** in GeMTest contains information such as the input data, or the test strategy specifying how to collect source inputs from the input data. Currently, GeMTest supports a *sampling* strategy which limits the number of MTCs to a specific value and an *exhaustive* strategy where all possible MTCs are generated and run. All registered MRs and generated MTCs are contained in a singleton **`MetamorphicTestSuite`** object.

IV. Metamorphic Relation Example

Listing 3 showcases a MR with two source inputs/outputs and two follow-up inputs/outputs. The relation accesses source inputs (Line 14), follow-up outputs (Line 15), and a randomly sampled parameter s (Line 16) to compute the boolean test verdict. This MR tests two integer list sorting SUTs (Line 21 and Line 25) the following: the MR receives two randomly generated source input lists (Line 8), and computes an element e that is at least one greater than the maximum of both lists (Line 9). Follow-up input one is the concatenation of both source input lists $(a + b)$, and follow-up input two is the same list, but with the large additional element e prepended to the beginning of this list $([e]+a+b)$ (Line 10). The relation checks whether this large element now correctly moved from the beginning of the list to the end of the list by checking if follow-up output one $+[e]$ equals follow-up output two (Line 17). This MT tests whether the sorting algorithms handle edge cases where an element must be moved completely through the list.

V. Test Result Analysis

For analyzing metamorphic test results, testers can use standard pytest outputs, such as command-line results or IDE-integrated reports. For example, executing Listing 1 produces the output shown in Listing 2. Additionally, GeMTest provides custom test result analysis via a database export, which can be visualized using the gemtest-webapp. This detailed logging is enabled by running pytest with the *--html-report* flag. As shown in Figure 2, the gemtest-webapp displays all MTCs from a test run, allowing test engineers to filter results and open the MTC detail view for further analysis. For example, Figure 3 shows the MTC detail view for a MTC that mirrors a traffic sign. This web-based technology is especially useful if data cannot be easily visualized on a command line, as it is the case for images.

VI. Validation Study

We validated GeMTest's functionality by implementing 218 MRs across 16 program domains. These exemplary MRs are publicly available in the gemtest-examples repository. To further assess GeMTest, we plan to conduct experiments comparing its performance overhead against a pure pytest-based MR implementation. Additionally, we aim to study the adoption of GeMTest by test engineers.

VII. Limitations and Future Work

We identify two main limitations of GeMTest: (1) While GeMTest can, in theory, test any program callable within Python – such as Java code executed via the command line or through tools like py4j – a native integration of MT into Java testing frameworks like JUnit [7] would be preferable. (2) GeMTest introduces performance overhead, particularly when testing DL systems. Reducing this overhead will be a focus of future work.

For future work, we propose implementing an automated composition of MRs, as suggested by Liu et al. [8]. This would increase the number of MRs and potentially uncover previously unknown defects. Additionally, we plan to integrate existing pytest-compatible tools, such as mutation testing frameworks, to compute mutation scores for specific MRs. Finally, we aim to explore the use of GeMTest as an advanced data augmentation technique in semi-supervised learning schemes, leveraging MT to fix defects in DL models.

VIII. Conclusion

GeMTest is a framework that automatically generates and executes general MTCs, as described by Chen et al. [3], from MRs implemented using Python decorators provided by GeMTest. Being domain-independent, GeMTest can be used to test classical as well as DL programs of any kind. Since GeMTest produces pytest test cases, it integrates seamlessly into existing pytest infrastructures and workflows. Our framework helps practitioners formulate MRs by providing an intuitive structure and implementing MT into their existing test suites. GeMTest is open-source and can be easily installed via the Python Package Index (PyPI) https://pypi.org/project/gemtest/.

Acknowledgment

We thank Paul Schwind, Martin Rau, Ting-Yu Lu, Danny Benlin Oswan, Tathagata Bandyopadhyay, Fabian Schlachter, Moritz Pötzsch, Robin Brase, Alexander Hobmeier, and Mino Estrella for their help in implementing GeMTest.

References

[1] Pablo C. Cañizares et al. “New ideas: Automated engineering of metamorphic testing environments for domain-specific languages”. In: *International Conference on Software Language Engineering, SLE*. Chicaco, IL, USA, 2021, pp. 49–54. DOI: 10.1145/3486608.3486904.

[2] Tsong Yueh Chen et al. *Metamorphic testing: A new approach for generating next test cases*. Tech. rep. The Hong Kong University of Science and Technology, 1998, p. 11. DOI: 10.48550/arXiv.2002.12543.

[3] Tsong Yueh Chen et al. “Metamorphic testing: A review of challenges and opportunities”. In: *ACM Computing Surveys* 51.1 (2018), pp. 1–27. DOI: 10.1145/3143561.

[4] Yao Deng et al. “A declarative metamorphic testing framework for autonomous driving”. In: *IEEE Transactions on Software Engineering* 14.8 (2022), pp. 1–20. DOI: 10.1109/tse.2022.3206427.

[5] *GeMTest Archive*. 2025. DOI: 10.6084/m9.figshare.28355363.

[6] Pablo Gómez-Abajo et al. “Automated engineering of domain-specific metamorphic testing environments”. In: *Information and Software Technology* 157.107164 (2023), p. 17. DOI: 10.1016/j.infsof.2023.107164.

[7] *JUnit*. [Online; accessed 09.09.2024]. 2024. URL: https://junit.org.

[8] Huai Liu et al. “A new method for constructing metamorphic relations”. In: *International Conference on Quality Software*. Xi'an, China: IEEE, 2012, pp. 59–68. DOI: 10.1109/QSIC.2012.10.

[9] Yelin Liu et al. “MTKeras: An automated metamorphic testing platform”. In: *International Journal of Software Engineering and Knowledge Engineering* 31.9 (2021), pp. 1235–1249. DOI: 10.1142/S021819402150039X.

[10] Yu Liu et al. “Inline tests”. In: *International Conference on Automated Software Engineering, ASE*. Rochester, MI, USA, 2022, p. 13. DOI: 10.1145/3551349.3556952.

[11] Yu Liu et al. “pytest-inline: An inline testing tool for python”. In: *International Conference on Software Engineering, ICSE*. Melbourne, Australia, 2023, pp. 161–164. DOI: 10.1109/ICSE-Companion58688.2023.00046.

[12] David MacIver and Zac Hatfield-Dodds. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (2019), p. 1891. DOI: 10.21105/joss.01891.

[13] Phu X. Mai et al. “SMRL: A metamorphic security testing tool for web systems”. In: *International Conference on Software Engineering, ICSE*. Seoul, Korea, 2020, pp. 9–12. DOI: 10.1145/3377812.3382152.

[14] Valentin J. M. Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[15] *pytest*. [Online; accessed 10.09.2024]. 2024. URL: https://pytest.org/.

[16] Sergio Segura et al. “A survey on metamorphic testing”. In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 805–824. DOI: 10.1109/TSE.2016.2532875.

[17] Simon Speth et al. “Safety-critical oracles for metamorphic testing of deep learning LiDAR point cloud object detectors”. In: *IEEE Open Journal of Intelligent Transportation Systems* 6 (2025), pp. 95–108. DOI: 10.1109/OJITS.2025.3532777.

[18] Elaine J Weyuker. “On testing non-testable programs”. In: *The Computer Journal* 25.4 (1982), pp. 465–470. DOI: 10.1093/comjnl/25.4.465.

[19] Yingzhuo Yang et al. “Towards effective metamorphic testing by algorithm stability for linear classification programs”. In: *Journal of Systems and Software* 180.111012 (2021), pp. 1–17. DOI: 10.1016/j.jss.2021.111012.

[20] Hong Zhu. “JFuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods”. In: *International Conference on Trustworthy Systems and Their Applications, TSA*. Hualien, Taiwan: IEEE, 2015, pp. 8–15. DOI: 10.1109/TSA.2015.13.