Chrysalis: A Lightweight Logging and Replay Framework for Metamorphic Testing in Python

Jai Parera*
University of California, Los Angeles
Los Angeles, CA, USA
jaiparera@cs.ucla.edu

Nathan Huey*
University of California, Los Angeles
Los Angeles, CA, USA
njhuey@g.ucla.edu

Ben Limpanukorn
University of California, Los Angeles
Los Angeles, CA, USA
blimpan@cs.ucla.edu

Miryung Kim
University of California, Los Angeles
Los Angeles, CA, USA
miryung@cs.ucla.edu

*Equal contribution.

***Abstract*—Metamorphic testing (MT) is a powerful technique for software testing. We introduce Chrysalis, a lightweight, extensible *logging and replay*-based metamorphic testing framework in Python. Chrysalis allows developers to define custom input transformations and their associated invariants, then execute structured metamorphic testing campaigns. Its key innovation is a lightweight logging mechanism that records the full history of transformations applied to an input. This compact representation enables developers to not only identify test failures but also to replay the exact sequence of transformations leading to a bug, facilitating debugging. We demonstrate Chrysalis's effectiveness through two case studies: auditing a machine learning model for fairness and assessing the robustness of large language models.**

A screencast demonstrating Chrysalis is available at: https://youtu.be/xJG4qghxIIs, and the source code is available at: https://github.com/Chrysalis-Test/Chrysalis.

I. Introduction

Software testing is often hampered by the "test oracle problem," where determining the correct output for a given input is difficult or impossible [1]. Metamorphic Testing (MT) offers a principled solution by verifying system properties, known as metamorphic relations (MRs), rather than specific input-output pairs. In this work, we define an MR to consist of a source input, a transformation to create a follow-up input, and an expected relationship between their corresponding outputs. For example, a web search for "software testing" should yield similar results to a search for "testing software," even if the exact "correct" set of results is unknown.

While MT has proven effective in research for testing everything from compilers and databases to machine learning models [2], its practical adoption remains low. A primary reason is the tooling gap: existing implementations are often domain-specific and are difficult to maintain, extend, and reuse. This results in a high barrier to entry and prevents the systematic application of MT in real-world software development workflows.

To address this challenge, we present *Chrysalis*, a general-purpose Python framework for metamorphic testing. It provides a general API to define transformations over the input and invariant properties over outputs before and after a transformation. Chrysalis automatically logs chains of input transformations, enabling developers to replay and reconstruct the inputs that led to violations of user-defined invariants.

The main contributions of this work are:

- We introduce Chrysalis, an extensible metamorphic testing framework for defining transformations and invariants.
- Chrysalis features a lightweight logging mechanism that stores metamorphic relation chains in a queryable database, enabling the ability to precisely replay failure-inducing transformation chains to analyze the conditions that trigger a bug.
- We demonstrate Chrysalis's utility in two domains: testing the fairness of prediction models and assessing the robustness of large language models.

II. Related Work

Metamorphic testing (MT) was originally introduced as a means to test software in domains where defining the expected output for a given input is intractable [1]. Various frameworks have been developed to apply MT to specific problem domains [2]. For instance, MT4I is a framework tailored for metamorphic testing of image processing algorithms [3]. Similarly, METAL provides a solution for testing large language models (LLMs). Unlike Chrysalis, both these tools are domain-specific and lacks a mechanism for replaying the history of transformations that lead to a failure, a key feature of our approach.

Among general-purpose frameworks, GeMTest shares the most similarities with Chrysalis [4]. Like Chrysalis, GeMTest is a general metamorphic testing framework that records test execution data into a database. However, Chrysalis also records the full sequence of transformations leading to a violation of an MR, enabling the exact replay and reconstruction of the transformations that lead to a failure. This is particularly

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00371

crucial for identifying bugs that only manifest after a specific series of state changes. Furthermore, Chrysalis employs a more lightweight logging strategy; it stores only the initial input and the chain of transformations. This enables Chrysalis to efficiently scale to test campaigns with long transformation chains.

III. Design and Methodology

Chrysalis introduces a structured, three-step process for metamorphic testing, designed to enable systematic and repeatable test campaigns.

A. Step 1: Define Metamorphic Relations

The developer begins by defining the metamorphic relations (MRs) relevant to their system under test (SUT). In Chrysalis, an MR is composed of a *transformation* defined over the input and one or more *invariants* defined over the outputs before and after the transformation.

A transformation is a pure Python function that takes an input object of type T and returns a modified object of the same type T.

```
def flip_gender(features: dict) -> dict:
    new_features = features.copy()
    new_features["gender"] = 1 - new_features["
    gender"]
    return new_features
```

Listing 1. Example of a transformation that flips the gender feature of the input.

An **invariant** is a predicate that compares the SUT's output from the original input with the output from the transformed input. It returns 'True' if the expected invariant holds.

```
# Example: Invariant checking for label stability
def label_invariant(out1: dict, out2: dict) -> bool:
    return out1["label"] == out2["label"]
```

Listing 2. Example of an invariant that checks label stability after a transformation.

These functions are registered with Chrysalis, which uses them to construct chains of metamorphic relations.

B. Step 2: Execute a Test Campaign

With a library of MRs defined, the user executes a test campaign against their SUT. Chrysalis automates the process of generating and running long sequences of transformations, which we call **transformation chains**.

```
chry.run(
    sut=my_income_predictor,
    input_data=test_samples,
    chain_length=10,
    num_chains=1000
)
```

Listing 3. Example of executing a test campaign with Chrysalis.

During execution, Chrysalis randomly selects an initial input from test_samples. It then constructs a chain by iteratively applying randomly selected transformations. After each transformation, it invokes the SUT on the new, modified input and evaluates all relevant invariants against the output of the previous step.

C. Step 3: Analyze Failures with Replayable Histories

The most significant contribution of Chrysalis is its approach to logging and analysis. Storing the full input and output at every step of a long transformation chain is computationally expensive and quickly becomes intractable. Chrysalis solves this with a lightweight logging strategy.

For each test run, Chrysalis stores only:

1) The **initial input object**.
2) The **sequence of transformations** applied (the "recipe").
3) A list of the **invariants that failed** at each step.

This information is stored in a structured SQLite database. Because transformations are required to be deterministic, this compact log contains all the information needed to perfectly reconstruct the exact input that caused any failure. A user can query the database for a specific failure (e.g., gender_bias_invariant violations) and use Chrysalis to *replay* the full transformation chain, allowing for interactive debugging and deep inspection of the SUT's state at the moment of failure.

IV. Case Study: Fairness in Income Prediction

To evaluate Chrysalis's ability to uncover subtle, unwanted behaviors in machine learning models, we conducted a case study using Chrysalis to test the fairness of an income prediction classifier. The goal was to demonstrate how Chrysalis can systematically probe for biases that are not apparent from standard accuracy metrics.

A. Setup

We used a logistic regression classifier trained on the UCI Adult Income dataset to predict if an individual's income exceeds \$50K/year. The model achieves ∼85% accuracy on a held-out test set. The SUT is a function that takes a dictionary of features and returns the predicted label and confidence score. Chrysalis was configured to execute 25 transformation chains of length 30 for each of the 2000 test inputs.

We define the following transformations that perturb the input features:

- 'flip_gender': This transformation toggles the "sex" field between the "Male" and "Female" categories.
- 'flip_race': This transformation toggles the "race" field across two selected racial categories.
- 'shift_age': This transformation increments or decrements the "age" field by a fixed amount.
- 'shift_education': This transformation modifies the "education-num" field to simulate a change in educational attainment.
- 'shift_hours': This transformation alters the "hours-per-week" field.
- 'tweak_capital_gain': This transformation perturbs the "capital-gain" feature within its valid bounds.

We also define the following invariants to check for unwanted model behaviors:

TABLE I
FAILURE RATES BY TRANSFORMATION AND INVARIANT (FAIRNESS CASE STUDY)

Transformation	Invariant	Failures	Total Pairs	Rate (%)
flip_gender	confidence	67555	242k	27.92
flip_gender	label	21832	242k	9.02
flip_gender	gender_bias	15895	242k	6.57
flip_race	race_bias	15442	272k	5.68
flip_race	label	4612	272k	1.70
shift_hours	label	4238	282k	1.50
shift_education	label	2668	184k	1.45
shift_education	confidence	2093	184k	1.14
shift_age	label	1271	246k	0.52
tweak_capgain	label	46	274k	0.02
shift_education	education_bias	14	184k	0.01

- 'label_invariant': This invariant asserts that the output label should remain the same across certain transformations.
- 'confidence_invariant': This invariant checks that the model's confidence score does not vary drastically due to semantically irrelevant changes.
- 'gender_bias_invariant': This invariant checks for significant changes in the prediction or confidence when the gender attribute is flipped.
- 'race_bias_invariant': This invariant is analogous to the gender bias check, but it is applied to the race attribute.
- 'education_bias_invariant': This invariant is used to identify excessive sensitivity in the model's output when the education field is modified.

B. Results

Table I summarizes the failure rates for the most salient transformation-invariant pairs. The 'flip_gender' transformation caused label changes over 9% of the time and large confidence shifts in nearly 28% of cases. In contrast, 'flip_race' was far more stable. This indicates that despite high overall accuracy, the model's predictions are disproportionately sensitive to the gender feature.

This case study shows how Chrysalis moves fairness testing from manual checks to a systematic, scalable process. More importantly, with the failure logs, a developer could isolate a specific 'gender_bias' failure and use Chrysalis to replay the exact chain of transformations (e.g., 'shift_age' $\rightarrow$ 'tweak_capital_gain' $\rightarrow$ 'flip_gender') that led to it, enabling targeted debugging of the model's behavior.

V. CASE STUDY: ROBUSTNESS OF LLM PARAPHRASING UNDER PROMPT PERTURBATION

This case study demonstrates Chrysalis's utility in the domain of Large Language Models (LLMs), where oracles are nonexistent and behavior can be highly stochastic. We evaluated the robustness of four different LLMs when tasked with rephrasing text under various prompt perturbations.

A. Setup

The SUT was a function that took a prompt and passed it to an LLM to generate a rephrased paragraph. We tested four

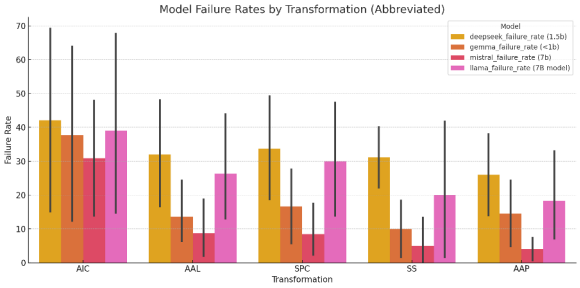

Fig. 1. Failure rate grouped by transformation

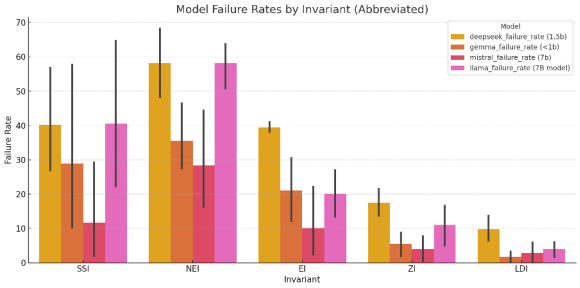

Fig. 2. Failure rate grouped by invariant

models: DeepSeek-R1 (1.5B) [5], Gemma3 [6] (<1B), Mistral (7B) [7], and LLaMA-3.2 (7B) [8], using prompts drawn from varied domains.

We defined five different transformations that perturb the input prompt:

- 'shuffle_prompt_clauses' (SPC): This transformation reorders the clauses within the sentences of the prompt.
- 'synonym_substitution' (SS): This transformation replaces a word in the prompt with a synonym sourced from the WordNet database [9].
- 'add_irrelevant_context' (AIC): This transformation prepends a true but topically unrelated sentence to the original prompt.
- 'ask_as_paragraph' (AAP): This transformation adds an instruction requesting that the output be formatted as a single paragraph.
- 'ask_as_list' (AAL): This transformation adds an instruction requesting that the output be formatted as a list.

We also define the following linguistic invariants which should be preserved after the transformations:

- 'named_entity_invariant' (NEI): This invariant checks that the Jaccard similarity of named entities between the original and transformed outputs is at least 0.6 [10].
- 'entropy_invariant' (EI): This invariant asserts that the information entropy of the output text does not change by more than ± 0.25 bits.
- 'lexical_diversity_invariant' (LDI): This invariant verifies that the lexical diversity, measured by the type-token ratio, remains stable within a threshold of ± 0.1 [11].

- 'zipf_invariant' (ZI): This invariant checks that the exponent of the Zipfian distribution of word frequencies does not change by more than ± 0.1 [12].
- 'semantic_similarity_invariant' (SSI): This invariant asserts that the semantic similarity between the outputs, measured by cosine similarity, remains at or above 0.9.

B. Results and Analysis

Chrysalis executed the test campaign and logged all invariant violations. The aggregated results, shown in Figures 1 and 2, reveal clear patterns in model behavior.

The results highlight significant transformation sensitivity, with the 'add_irrelevant_context' (AIC) transformation causing the most failures across all models. These failures were particularly pronounced for the 'semantic_similarity_invariant' (SSI) and 'named_entity_invariant' (NEI), suggesting that distracting context degrades a model's ability to preserve core meaning. The 'shuffle_prompt_clauses' (SPC) transformation also induced notable instability, albeit to a lesser extent. In terms of invariant difficulty, NEI and SSI were the most frequently violated invariants overall, indicating a common struggle among the models to consistently maintain named entities and semantic coherence under perturbation. Finally, our model comparison revealed that Mistral (7B) was the most robust model, while DeepSeek and Gemma showed higher fragility across most tests.

These findings support the utility of Chrysalis for comparative robustness testing. Rather than hand-crafting unit tests, the system applies automated transformation chains and invariant checks to uncover fragile behaviors at scale.

VI. Conclusion

In this work, we introduced Chrysalis, a general framework for metamorphic testing in Python that provides a novel logging and replay mechanism that enables users to precisely reconstruct the series of input transformations that violate a user-defined invariant. Our case studies in ML fairness and LLM robustness demonstrate its ability to systematically uncover biases and unwanted behavior in domains with limited or no explicit test oracles.

Chrysalis is open-source and we hope it will serve as both a valuable research platform and a practical asset for building more reliable software.

Chrysalis is available at: https://github.com/Chrysalis-Test/Chrysalis.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, and 2106404. It is also supported in part by funding from Amazon and Samsung. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

References

[1] T. Y. Chen, S. C. Cheung, and S. M. Yiu, *Metamorphic testing: A new approach for generating next test cases*, 2020. arXiv: 2002.12543 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2002.12543.

[2] S. Segura, G. Fraser, A. B. Sánchez, *et al.*, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, pp. 805–824, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:16426065.

[3] C.-A. Sun, J. Xing, X. Li, *et al.*, "Metamorphic testing of image processing applications: A general framework and optimization strategies," in *Proceedings of the 9th ACM International Workshop on Metamorphic Testing*, ser. MET 2024, Vienna, Austria: Association for Computing Machinery, 2024, pp. 26–33, ISBN: 9798400711176. DOI: 10.1145/3679006.3685070. [Online]. Available: https://doi.org/10.1145/3679006.3685070.

[4] S. Speth and A. Pretschner, "GeMTest: A General Metamorphic Testing Framework," in *Proceedings of the 47th International Conference on Software Engineering, (ICSE-Companion)*, Ottawa, ON, Canada, 2025, pp. 1–4.

[5] DeepSeek-AI, D. Guo, D. Yang, *et al.*, *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[6] G. Team, T. Mesnard, C. Hardin, *et al.*, *Gemma: Open models based on gemini research and technology*, 2024. arXiv: 2403.08295 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2403.08295.

[7] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2310.06825.

[8] A. Grattafiori, A. Dubey, A. Jauhri, *et al.*, *The llama 3 herd of models*, 2024. arXiv: 2407.21783 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2407.21783.

[9] Princeton University, *About WordNet*, https://wordnet.princeton.edu, Accessed: 2025-07-23, 2010.

[10] R. Real and J. M. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic Biology*, vol. 45, no. 3, pp. 380–385, Sep. 1996, ISSN: 1063-5157. DOI: 10.1093/sysbio/45.3.380. eprint: https://academic.oup.com/sysbio/article-pdf/45/3/380/19501760/45-3-380.pdf. [Online]. Available: https://doi.org/10.1093/sysbio/45.3.380.

[11] F. Tweedie and H. Baayen, "How variable may a constant be? measures of lexical richness in perspective," *Computers and the Humanities*, vol. 32, pp. 323–352, Sep. 1998. DOI: 10.1023/A:1001749303137.

[12] S. T. Piantadosi, "Zipf's word frequency law in natural language: A critical review and future directions," *Psychonomic Bulletin & Review*, vol. 21, no. 5, pp. 1112–1130, 2014. DOI: 10.3758/s13423-014-0585-6.