Metamorphic Testing of Deep Reinforcement Learning Agents with MDPMORPH

Jiapeng Li*, Zheng Zheng*, Yuning Xing†, Daixu Ren*, Steven Cho† and Valerio Terragni†
*Beihang University, Beijing, China
Email: {jp_li, zhengz, rendaixu}@buaa.edu.cn
†University of Auckland, Auckland, New Zealand
Email: {yxin683, scho518, vter674}@aucklanduni.ac.nz

***Abstract*—We present MDPMORPH, a tool for metamorphic testing of Deep Reinforcement Learning (DRL) agents. MDPMORPH is based on the Markov Decision Process (MDP) and targets the core reasoning properties of DRL agents to automatically uncover potential faults. It can generate metamorphic test suites and corresponding mutants directly from the DRL system under test. MDPMORPH uses a subset of the metamorphic test suite and models to train the thresholds of the nine proposed Metamorphic Relations (MRs) using stochastic gradient descent. These MRs are based on the temporal characteristics of the MDP, and the training aims to determine the optimal threshold for each MR. After obtaining the optimal threshold, MDPMORPH leverages the MRs to compare the execution results of different metamorphic test suites on the model under test and reports whether each test passes or fails. Finally, by collecting the execution results, MDPMORPH calculates the mutant detection rate of MR to validate its effectiveness. Experimental results show that MDPMORPH and the proposed MRs are highly effective in automatically detecting seeded faults (mutants).**

***Index Terms*—SE4AI, deep reinforcement learning, metamorphic testing, mutation testing**

I. INTRODUCTION

Deep Reinforcement Learning (DRL) agents have demonstrated remarkable performance across a variety of domains, including autonomous driving [1], [2], and large language model [3], [4]. Their ability to learn from high-dimensional observations and adapt to complex systems has led to their rapid adoption in both academic and industrial settings [5]. Therefore, to prevent catastrophic failures in real-world applications, safety-critical DRL agents must be extensively tested against various edge and failure scenarios to expose and resolve latent weaknesses. Recent efforts in testing DRL agents have explored various methods such as fuzzy testing [6], search-based testing [7], and mutation testing [8], [9]. While these techniques have advanced the state of DRL testing, they still rely heavily on human defined oracles, leaving the oracle problem a persistent challenge for automated verification.

Metamorphic Testing (MT) [10], as a promising approach, has demonstrated significant effectiveness in alleviating the oracle problem. By leveraging domain-specific Metamorphic Relations (MRs), which are properties that define how the output of the system should change or remain invariant under certain input transformations, MT enables automated, oracle-free validation of DRL agents. Such relations provide a foundation for detecting erroneous or unstable behavior without requiring labeled ground truth. However, effectively applying MT to DRL systems presents its own set of challenges. These include defining effective MRs suitable for agent reasoning, choosing appropriate output comparison metrics, and handling the stochasticity inherent in DRL systems. Furthermore, as DRL agents learn from dynamic environments, testing should assess both single-step reasoning and multi-step trajectories.

We have addressed the above challenges by proposing MDPMORPH [11], a novel MT framework for testing DRL agents, recently accepted to the main research track of ISSRE 2025. Our framework is grounded in the principles of Markov Decision Processes (MDPs) and targets the core reasoning behaviors of DRL agents to automatically uncover potential faults. To support MDPMORPH, we proposed a systematic methodology, based on MDP characteristics, for designing MRs tailored to DRL agents. Using this approach, we defined nine generic MRs that capture commonly expected properties of agent decision-making. We theoretically justified the soundness of these MRs using formal MDP assumptions and definitions.

This paper extends our recent work [11] with details about the design, implementation, and usage of MDPMORPH. The source code is publicly available on `https://github.com/tissten/MDPMorph`, and a demo is available on `https://youtu.be/IwMaUfQ2xnQ`.

MDPMORPH is a tool capable of automatically testing DRL systems through metamorphic testing. It extracts initial states from the environment as source test cases, generates metamorphic test pairs via input transformations, and creates mutants by applying mutation rules. MDPMORPH tunes MR sensitivity via threshold training, executes test pairs to identify behavioral inconsistencies, and evaluates the effectiveness of each MR based on the observed results. The envisioned users of MDPMORPH are machine learning engineers and researchers working with DRL. MDPMORPH is designed to help engineers automatically test their DRL models using metamorphic testing. It also provides researchers with the flexibility to define and experiment with new MRs.

Corresponding authors: Zheng Zheng (zhengz@buaa.edu.cn) and Valerio Terragni (v.terragni@auckland.ac.nz)

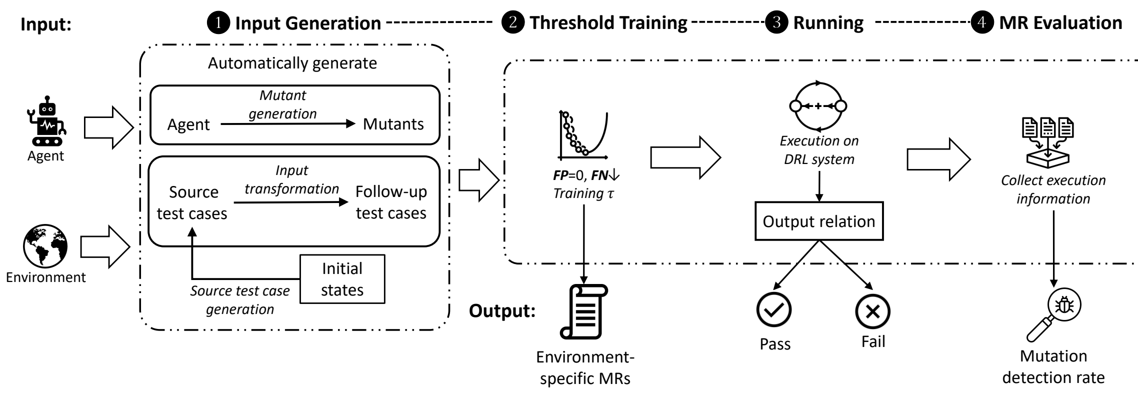

Figure 1. Logical workflow of MDPMORPH

II. MDPMORPH

MDPMORPH is based on MDPs and focuses on the agent's core reasoning properties to automatically uncover faults. Figure 1 illustrates its logical workflow. The MDPMORPH comprises four components: *Input Generation*, *Threshold Training*, *Running* and *MR Evaluation*.

Inputs. The inputs of MDPMORPH are agent and environment, where the agent is used for mutant generation and the initial states from the environment are extracted as test cases.

Outputs. In MDPMORPH, different components have different outputs. Specifically, 1) the output of the *Threshold Training* is a set of environment-specific MRs with optimized thresholds. 2) the output of the *Running* is the execution result (pass or not). 3) the output of the *MR Evaluation* is the mutation detection rate.

How MDPMORPH works. Currently, MDPMORPH implements nine MRs [11] grounded in the sequential properties of MDPs. They are designed to capture variations in key aspects of DRL systems, such as states, actions, and rewards, while examining action outputs, cumulative rewards, and decision sequence stability. Considering the inherent stochasticity of DRL systems, we set threshold for each MR to maximize adaptability across different systems, acknowledging that these thresholds may vary depending on the specific DRL system. We use the original agent and its mutants to search for the optimal thresholds for the MRs and to evaluate their effectiveness. It is worth noting that, under certain assumptions, we provide theoretical proofs of the necessary properties of these MRs. If the DRL system does not satisfy these assumptions, users can still obtain optimal thresholds for the MRs using the *Threshold Training* component provided by MDPMORPH, but this may potentially lead to false positives. In addition, MDPMORPH is designed with flexibility in mind. During its usage, users are not limited to the predefined set of MRs. Instead, they are encouraged to define and incorporate new MRs that align with the characteristics of their specific domains, environments, or agent behaviors.

The following example illustrates how MDPMORPH detects the faulty behavior of a mutant. In our experiments, a representative violation discovered by an MR occurred in the CARTPOLE system using MR4 [11]. The CARTPOLE system simulates a classic control problem where the goal is to balance a pole on a moving cart by applying forces to the cart [12]. This MR mandates similar decision-making when source and follow-up test cases are identical, ensuring differences between resulting trajectories remain minimal. However, mutant "NEB_03" violated this MR: the mutant successfully maintained pole balance in the source test case but consistently pushed the cart incorrectly in the follow-up test case. Consequently, the pole rapidly fell, terminating the task prematurely. This clearly demonstrates an unreasonable decision by the mutant, highlighting its vulnerability to minor environmental perturbations and indicating a clear flaw in its decision making policy.

III. TOOL USAGE

Implementation Details. MDPMORPH is implemented in PYTHON. It leverages the GYMNASIUM[1] library and the CARES[2] platform to construct an automated metamorphic testing tool for testing and evaluating DRL agents.

Installation instructions. MDPMORPH requires Python 3.10 for compatibility. All necessary dependencies can be installed via `pip install -r requirements.txt`.

In a nutshell, MDPMORPH has four main components:

① **Input Generation.** MDPMORPH generates source test cases by randomly sampling from the range of initial states defined during the training process of the agent under test, and applies input transformations to generate follow-up test cases, thereby forming metamorphic test pairs. Meanwhile, it generates mutated agents by applying predefined mutation rules to the original agent. These mutation rules are derived from the model-level mutation rules in DEEPMUTATION [13]. These metamorphic test pairs and mutants are generated as output and used in subsequent components.

② **Threshold Training.** MDPMORPH continuously adjusts the thresholds in generic MRs by minimizing False Nega-

[1]https://gymnasium.farama.org
[2]https://github.com/UoA-CARES/cares_reinforcement_learning

tives (FNs) while ensuring zero False Positives (FPs) [14], [15], thereby generating environment-specific MRs as output.

③ **Running.** MDPMORPH executes metamorphic test pairs on the model and compares their outputs to evaluate whether the expected output relation defined by each MR holds, generating execution results as output.

④ **MR Evaluation.** MDPMORPH collects execution results and outputs the mutation detection rate to evaluate the effectiveness of the MR.

To support modularity, each component can run independently. If needed, users can easily create a simple script to invoke all components in a pipeline. The following subsections describe how to use MDPMORPH.

A. Input Generation

The *Input Generation* component of MDPMORPH is responsible for automatically generating metamorphic test pairs and mutants, which serve as inputs to the subsequent components.

First, run `generate_tests.py` to randomly sample valid states from the state space and generate the specified number of source test cases and test suites. For example, for the BIPEDALWALKER system, MDPMORPH randomly samples from 13 dimensions of the state space. Then, based on the input transformation defined in `MR.py`, the source test case is transformed into a follow-up test case, forming a metamorphic test pair. By running different mutant generation files, a specified number of mutants can be generated by mutating the trained agent. For example, running `Gaussian_Fuzzing_operator.py` generates mutants related to weight fuzzing.

B. Threshold Training

The goal of the *Threshold Training* component is to determine the optimal thresholds for the nine predefined MRs. To achieve this, MDPMORPH uses the execution results of metamorphic test pairs on both the original agent and its mutants to train the thresholds in predefined MRs. Specifically, run `run_threshold_training.py`, providing the path to the model to be used and the path to the MR whose threshold needs to be trained. During the threshold training process, we save the threshold values at two levels for detailed observation:

- **Test case level.** After executing each test case, the updated threshold value is recorded in the `threshold_training_test_case.csv` file.
- **Test suite level.** After completing the execution of each full test suite, the updated threshold value is recorded in the `threshold_training_test_suite.csv` file.

Once the threshold converges, meaning it remains stable over several evaluation iterations, the resulting value is selected as the optimized threshold for the environment-specific MR.

It should be noted that threshold training in our tool is an optional component, allowing users to specify custom thresholds. The thresholds obtained after training can only be used in regression testing mode to detect faults in future versions of the agent.

C. Running

The main goal of the *Running* component is to execute the metamorphic test pairs and verify the results to determine whether the predefined MRs are violated. Run the `run.py` file and provide the path to the pre-trained model along with the path to the MR to be evaluated. The tool then determines pass/fail outcomes by checking whether the output relation defined by the MR is satisfied.

In our current implementation, MDPMORPH supports the following types of output relations:

- **Trajectory distance**: Compute the total step-by-step Euclidean distance between state sequences.
- **Cumulative reward difference**: Compute the difference in cumulative rewards between the source and follow-up test cases.
- **Action distribution divergence**: Compute the difference between the action probability distributions output by the policy under similar test cases before and after transformation.

MDPMORPH also allows users to select the appropriate comparison metric based on the definition of the MR.

In addition, if the user needs to add new MRs, they only need to provide the path to store the corresponding MRs when running the `run.py` file. Note that MRs must implement essential system functions such as `env.reset()`, `env.step()`, `agent.select_action()`, and others.

By default, each metamorphic test pair is executed only once. Meanwhile, MDPMORPH retains support for repeated executions, enabling users to optionally activate multiple runs when necessary to suit broader application scenarios. The number of repeated executions can be configured by modifying the `number_eval_episodes` parameter in the `train_config.json` file.

D. MR Evaluation

To evaluate the effectiveness of MRs, the *MR Evaluation* component gathers the execution results of metamorphic test pairs on both the original agent and its mutants, produced by the *Running* component, and calculating the mutation detection rate of each MR. It is worth noting that to ensure a fair evaluation, the *MR Evaluation* component uses a different set of metamorphic test pairs and mutants from those used in *Threshold Training*.

After the execution of `run.py`, the tool automatically generates a file that records the violations of a specific MR by a particular mutant across multiple test suites during execution. The file is named accordingly (e.g., `MR3_1_output_WS_3.csv`). These files record two key metrics:

- **Violation rate**: the percentage of test cases within a test suite that violate the expected MR.
- **Input**: the number of executed test cases in the suite.

This component is also optional and not mandatory, as it enhances confidence in the tool's results and model's robustness, especially when high mutation scores are achieved despite no detected faults.

Table I
AVERAGE MUTATION DETECTION RATES PER MR (SEE DEF. IN [11])

MR	**CARTPOLE**	**LUNARLANDER**	**BIPEDALWALKER**
MR1.1	0.77	0.88	0.94
MR1.2	0.77	0.84	0.79
MR1.3	0.74	0.86	0.88
MR2	0.68	0.96	0.70
MR3.1	0.59	1.00	0.91
MR3.2	0.63	1.00	0.90
MR3.3	0.54	1.00	0.91
MR4	0.87	0.83	0.80
MR5	0.96	0.97	0.96

IV. EVALUATION

We evaluated MDPMORPH on three environments: CARTPOLE, LUNARLANDER, and BIPEDALWALKER, all provided by the OpenAI GYMNASIUM. For each environment, we conducted threshold training and evaluation of the nine MRs using 50 test suites (40 for threshold training and 10 for evaluation), where each test suite consists of 100 distinct test cases, and 160 mutants (80 for threshold training and 80 for evaluation). These mutants were generated using model-level mutation operators provided by DEEPMUTATION [13]. Table I shows the average mutant detection rate for each MR. We found that under these MRs, all mutants were detected at least once, with an average detection rate of 0.84, demonstrating strong performance in complex systems. In addition, we analyzed the characteristics of undetected mutants under each MR. For more information about the experimental setup and results, please refer to our ISSRE paper [11].

V. RELATED WORK

Testing for Deep Reinforcement Learning (DRL) systems primarily involves search-based testing [7] and fuzzing techniques [6]. Similarly, MDPMORPH also uses MDP models, takes initial states as test inputs, and shares some of the same assumptions and the general goal of exposing bugs in DRL agents. However, these techniques all rely on manually crafted test oracles. In contrast, MDPMORPH employs MT to automate the oracle problem, thereby eliminating the need for and cost of manual oracle construction.

The work most closely related to ours is the study by Eniser et al. [16], who proposed a relaxed MR to detect decision inconsistencies under environment variations, using MT to validate action policies in RL system. Their MR focuses solely on tolerance based policy deviations, whereas MDPMORPH incorporates nine generic MRs spanning multiple semantic levels. In addition, MDPMORPH introduces threshold training for automatic MR instantiation, and supports end-to-end automation from test generation to mutant evaluation.

VI. CONCLUSION

While there are various testing methods for DRL systems, Metamorphic Testing (MT) in this domain remains in its early stages. MDPMORPH is among the first techniques to explore MT for DRL, pioneering research in this emerging area. Techniques like MDPMORPH can encourage developers to adopt more rigorous testing practices, ultimately improving the quality and reliability of DRL systems over time. We identify three promising future research directions: (i) enhancing automated source test case generation to better cover complex behaviors, (ii) developing richer and more varied MRs to capture subtle faults, and (iii) diversifying mutation strategies to capture a wider range of erroneous behaviors.

ACKNOWLEDGEMENTS

This work was supported in part by the National Key R&D Program of China under Grant 2024YFB3311503, and the National Natural Science Foundation of China under Grant 62372021.

REFERENCES

[1] Z. Huang, J. Wu, and C. Lv, "Efficient deep reinforcement learning with imitative expert priors for autonomous driving," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 10, pp. 7391–7403, 2022.
[2] C.-J. Hoel, K. Wolff, and L. Laine, "Ensemble quantile networks: Uncertainty-aware reinforcement learning with applications in autonomous driving," *IEEE Trans. Intell. Transp. Syst.*, vol. 24, no. 6, pp. 6030–6041, 2023.
[3] M. Hariharan, "Reinforcement learning: Advanced techniques for llm behavior optimization," *ESP Int. J. Adv. Comput. Technol.*, vol. 2, no. 2, pp. 84–101, 2025.
[4] K. Liu, D. Yang, Z. Qian, W. Yin, Y. Wang, H. Li, J. Liu, P. Zhai, Y. Liu, and L. Zhang, "Reinforcement learning meets large language models: A survey of advancements and applications across the llm lifecycle," *arXiv preprint arXiv:2509.16679*, 2025.
[5] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.
[6] Q. Pang, Y. Yuan, and S. Wang, "MDPFuzz: testing models solving markov decision processes," in *ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 378–390.
[7] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh *et al.*, "A search-based testing approach for deep reinforcement learning agents," *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3715–3735, 2023.
[8] D.-G. Thomas, M. Biagiola, N. Humbatova, M. Wardat, G. Jahangirova, H. Rajan, and P. Tonella, "μPRL: A mutation testing pipeline for deep reinforcement learning based on real faults," in *IEEE/ACM Int. Conf. Softw. Eng.*, 2025, pp. 2238–2250.
[9] J. Li, Z. Zheng, X. Du, H. Wang, and Y. Liu, "DRLMutation: A comprehensive framework for mutation testing in deep reinforcement learning systems," *ACM Trans. Softw. Eng. Methodol.*, 2025.
[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *HKUST Technical Report*, 1998.
[11] J. Li, Z. Zheng, Y. Xing, D. Ren, S. Cho, and V. Terragni, "MDPMORPH: an MDP-based metamorphic testing framework for deep reinforcement learning agents," in *IEEE Int. Symp. Softw. Reliab. Eng.*, 2025.
[12] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst., Man, Cybern.*, no. 5, pp. 834–846, 2012.
[13] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "DeepMutation: mutation testing of deep learning systems," in *IEEE Int. Symp. Softw. Reliab. Eng.*, 2018, pp. 100–111.
[14] J. Ayerdi, V. Terragni, G. Jahangirova, A. Arrieta, and P. Tonella, "GenMorph: automatically generating metamorphic relations via genetic programming," *IEEE Trans. Softw. Eng.*, vol. 50, no. 7, pp. 1888–1900, 2024.
[15] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *ACM Int. Sympo. on the Found. of Soft. Eng.*, 2020, pp. 1178–1189.
[16] H. F. Eniser, T. P. Gros, V. Wüstholz, J. Hoffmann, and M. Christakis, "Metamorphic relations via relaxations: An approach to obtain oracles for action-policy testing," in *ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 52–63.