2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678706

Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations

Leonhard Applis, Annibale Panichella, Arie van Deursen
Technische Universiteit Delft, Netherlands
{L.H.Applis, A.Panichella, Arie.vanDeursen}@tudelft.nl

***Abstract*—Metamorphic testing is a well-established testing technique that has been successfully applied in various domains, including testing deep learning models to assess their robustness against data noise or malicious input. Currently, metamorphic testing approaches for machine learning (ML) models focused on image processing and object recognition tasks. Hence, these approaches cannot be applied to ML targeting program analysis tasks. In this paper, we extend metamorphic testing approaches for ML models targeting software programs. We present LAMPION, a novel testing framework that applies (semantics preserving) metamorphic transformations on the test datasets. LAMPION produces new code snippets equivalent to the original test set but different in their identifiers or syntactic structure. We evaluate LAMPION against CodeBERT, a state-of-the-art ML model for Code-To-Text tasks that creates Javadoc summaries for given Java methods. Our results show that simple transformations significantly impact the target model behavior, providing additional information on the models reasoning apart from the classic performance metric.**

I. Introduction

Artificial Intelligence (**AI**) has been applied to software engineering (**SE**) to address many tasks, such as fault localization [1], test-case generation [2], fuzzing [3] or optimizing meta-parameters [4]. Recently, modern sequence-to-sequence deep learning models have shown promising results sparking new types of applications. Among them is the creation of code from verbatim description (*tex-to-code*) [5], or generation of documentation for source-code of previously unseen methods (*code-to-text*) [6, 7]. Yet, we argue that *it is not clear the extent to which these models truly behave as intended*, apart from their reported accuracy. Hence, applying testing strategies for ML-based program analysis solutions is critical.

In recent years, there has been great interest in Testing ML, where the goal is indeed to go beyond assessing accuracy (see the survey by Zhang et al. [8]). Many of the approaches have been taken from *classic* software testing and have been adapted for ML. One example is metamorphic testing, which is a well-established technique that is considered a powerful approach as it addresses the *Oracle Problem* [9] in test generation. Metamorphic testing has been successfully used in ML [10, 11] for image processing and object recognition. For example, image rotation is an information-preserving transformation as it alters the pixels in the image without changing its label (oracle). In computer vision, a *robust* ML model must not provide different predictions for the image altered with metamorphic transformations. Hence, quantifying the number of transformed images on which an ML model provides different answers quantifies its *robustness* against different transformations.

While extensive research has been conducted on metamorphic testing for vision computing tasks [10–12], the existing metamorphic transformations are domain-specific. Consequently, they cannot be applied and do not hold for different domains and types of data. In this paper, we *extend the concept of metamorphic testing to machine learning models trained on and targeting source code*.

We define a set of transformations that alter features of code but yield the effectively equal program, such as introducing `if(true)`-conditions or `+0` behind integer expressions. Using those, we modify the test-datapoints (programs) in order to detect differences in the models' predictions and metrics. We expect that the models are robust towards some transformations while others affect the metrics (negatively). The information gained could help to evaluate existing models, compare them to each other and provide suggestions and warnings for end-users and researchers alike. With our research and tool, we contribute on the following points:

1) Create a systematic approach, namely LAMPION, to quantify the robustness of a source-code-based model
2) Enable researchers to compare the *robustness* of models similarly to existing quality metrics
3) Groundwork for data augmentation in the field of ML4SE.
4) Empirically show the importance of robustness and testing when referring to ML-based program analysis.

To the best of our knowledge, we are the first to propose the use of metamorphic transformations for assessing ML-based program analysis tools. Our initial experiments on CodeBERT [5], a state-of-the-art ML model widely used in the SE literature [13–16], demonstrate the feasibility of the approach, and the type of lessons that can be learned from applying LAMPION.

II. Background and Related Work

Metamorphic testing is a technique based upon the concept of *metamorphic relations*, which is a property-based technique that exploits known equality of certain output values. Prominent examples are programs that implement mathematical functions; The sine function has a well-known metamorphic relation: $\forall x \in \mathbb{R} : sin(x) = sin(x + 2\pi)$. Testers can easily create new test cases based on this relation and assess the program

DOI 10.1109/ASE51524.2021.00172

correctness. A broad view of metamorphic testing studies and applications can be found in the survey by Segura et al. [17]. While metamorphic testing has not been applied to ML models for SE, metamorphic transformations and relations are known in software engineering and are tightly coupled to *refactoring*, program *optimization*, and linting. Metamorphic transformations are also used for compiler optimization to create more efficient code, using techniques like loop unrolling or function inlining [18].

Metamorphic Testing for ML. Metamorphic testing has been applied recently to machine learning, especially to image-based object-detection tasks [11][10]. A metamorphic transformation on images performs *information-preserving* alternations on an image. For example, the image of a cat might be mirrored, yet a classifier should still be able to recognize it as such. Blurring or saturating of images [19] change the data significantly; nevertheless, they are still easily classifiable by humans. These transformed images can be used to access robustness by generating more datapoints in the test set [11]. It can also be applied to generate more training data, which can result in a more robust or precise model [12].

The existing literature focuses on MTs that are specific to images and pixels. In this paper, we transplant the testing methodology to a new domain, namely ML models designed for program analysis. This requires defining new metamorphic relations and transformations for our domain, which we describe in Section III.

Adversarial attacks. Related work stems from Compton et al. [20] that introduces randomization of variable-names in the training dataset of a code2vec model for training data augmentation. Their study shows that the model trained on the augmented training dataset achieves slightly better accuracy than the model trained on the original dataset which motivates to systematically investigate for overfitting. Similarly, Yefet et al. [21] prove that they can generate adversarial attacks on Code2Vec-based classifiers by changing variable names or introducing new variables. As this existing research motivates to inspect identifier names, we include them into our approach in addition to other transformations.

III. Our Framework: Lampion

Overview. Figure 1 depicts the metamorphic testing approach, we named Lampion, and designed for testing ML models trained on source-code programs. Lampion relies on the MTs defined in the subsections below. Our approach consists three main steps. First, Lampion takes as inputs a pre-trained model and a program not used during the training process (items ⑤ and ① in Figure 1). It generates program variants (item ④) by using the MTs (item ②) and based on a given configuration file (item ③). The configuration specifies the type of transformation applied and the number of repetitions (order). Then, the original program and its equivalent variants are fed to the pre-trained model. Finally, Lampion compares the outcome produced by the pre-trained model for the original program (item ⑥) and its metamorphic variants (item ⑦). If there is no difference in the outcome, it means that the model is *robust* to the MT. Otherwise, we found a weakness in the model.

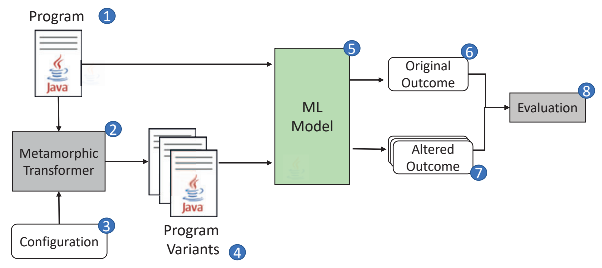

Figure 1. Lampion— Metamorphic Testing Framework for ML-based Program Analysis

Metamorphic Relation for Programs. The first step is to identify *metamorphic relations* for software programs, which are the data points for ML-based program analysis. *Metamorphic relations* (MRs) relate multiple programs that differ in their structures (e.g., AST) but that are effectively equivalent. As such, ML models should provide the very same output (e.g., same label) for programs that are related to one other according to an MR. Therefore, given a program P, we use MRs to generate equivalent yet different programs P'_1, ..., P'_k to test a given ML model under analysis.

In ML applications, the oracle function corresponds to the labels that humans provide for a given program P. The type of label for each program (data point) is task-dependent. For example, in ML-based program documentation, the label (oracle) is the natural language description developers write for the program P.

We identify two types of metamorphic relations for programs which are useful to test ML models for program analysis:

MR-1: *Addition of uninformative code elements*. Such a code element (e.g., comments, un-used variables, etc.) does not change the behavior of the target program P. As such, the label (oracle) for P and its variants with MR-1 relation remains the same.

MR-2: *Replace a code element with another equivalent element*. Equivalent program elements (e.g., different variable names) do not change the AST of the programs but the labels of the nodes within the AST. Using different yet equivalent elements does not change the behaviors of a program P either.

Metamorphic Transformations. Given the two MRs defined above, we can define a set of *metamorphic transformations* that satisfy our MRs. A *metamorphic transformation* (MT) is a procedure that generates new programs P'_1, ..., P'_k (*follow-up* programs) starting from an input program P and using a metamorphic relation. We have two constraints for MTs: First, the oracle function must give the same output for the initial program P and the transformed program $f(P)$. Second, P is a valid input for the ML model, then $f(P)$ must be valid input for the model too.

Table I
OVERVIEW OF METAMORPHIC TRANSFORMATIONS FOR PROGRAMS

Transformation	Short	Description	Estimated Effect	Variations
if-true	**MT-IF**	Wrapping a random expression in an *if(true)* statement	Structural Changes, introduction of conditions, introduction of keywords	if-false-else
add-unused-variable	**MT-UV**	Add a random unused variable	Introduction of names, introduction of types	Full random and pseudo random names (Postfix **R** & **P**), names looked up from a dictionary or the program under test
rename-entity	**MT-RE**	Rename a class, method or variable	Introduction of names, removal of known names	For variables, classes and member-types separate
lambda-identity	**MT-ID**	Wrap an expression in an identity-lambda function (including function call)	Introduction of complex structure, introduction of operators	-
delegation-method	**MT-DM**	extract an expression to a function, invoke the function instead of the method	Structural changes, change of scope for information, introduction of names	same as MT-UV
comment-alternation	**MT-CO**	Add,remove or move comments	Introduction or removal of natural language	Full or pseudo random comments generated
parameter-introduction	**MT-PI**	Introduce an unused parameter	Change of method signature, introduction of names, introduction of types	same as MT-UV
whitespace-alternation	**MT-WS**	Add or remove whitespace	Change of code-layout	-
add-neutral-element	**MT-NE**	Add the neutral element to a primitively typed expression	Change of structure, introduction of tokens	Complex equivalent transformations (e.g. replacing *true* with $0^1 == 1$)

A summary of MTs is presented in Table I. They target various features of the code, such as structure, tokens, and identifier names. Different models are known to have constraints by their design. For example, Code2Vec defines an AST-depth; hence, the model is known to break when introducing many redundant structure elements. Other models — especially deep learning models like CodeBERT — do not specify the features they target and were not previously inspected.

The presented table can be considered a starting point for metamorphic transformations applied to ML-based program analysis solutions.

IV. EMPIRICAL STUDY

We first want to assess whether the proposed MTs impact the performance of machine learning models. In an ideal case, ML models should not be affected by the metamorphic transformations, i.e., the model is not sensitive to changes that do not alter the code behavior. Hence, RQ1 should cover the general impact of applying one single transformation at the time, hereafter referred to as *first-order* MTs:

Research Question 1
To what extent do metamorphic transformations affect the performance of ML models?

We also want to compare the different types of transformations w.r.t the benchmark. We may expect that different transformations have different impacts on ML models. Furthermore, we aim to understand which model features are more robust, e.g., whether name-changes affect the model *more* than structural AST changes.

Research Question 2
To what extent do different types of MTs have a different impact on the performance of ML models?

Benchmark. For an initial study, we picked CodeBERT [5], particularly its downstream task of code-summarization. Code-summarization should clarify *what the model understands*, and the output can give clearer insights than *cold metrics*. We trained a CodeBERT-Java Model as described in the official repository by Microsoft [22], using the standard parameters given in the readme. CodeBERT has been trained on 6 programming languages with a total of 8.3M datapoints (code snippets) and achieves state-of-the-art results of an average BLEU4-Score of 17.65 in the CodeSearchNet-challenge [23].

Methodology / Experiment Design. We developed a metamorphic transformer for Java-Programs that works at the source-code level. In addition, we need a (pretrained) model and an existing benchmark that either consists of .java files or provides sufficient pre- and post-processing to transform the datapoints. **To answer RQ1**, we apply MTs to all datapoints in the test set, resulting in a set of variant-code-snippets. We then re-calculate the performance metrics for the variant-code snippets (metamorphic test cases) as well as for the original ones. We use the BLEU4-Score [24] as performance metric, which is the standard metric used in code-to-text and text-to-code generation tasks [16, 24]. The BLEU4-score is computed by tokenizing the gold standard and the generated text into n-grams, and comparing the resulting sets of n-grams. The metric value ranges from 1.0 (perfect translation) to 0 (not a single matching word or n-gram). In addition, we use the Jaccard-distance to measure the percentage of words that differ between the two Java-doc-comments generated by CodeBERT before and after applying an MT. To assess the significance of the differences in BLEU4-score achieved by the model with and without metamorphic transformations, we use the Wilcoxon rank-sum test [25]. We verified beforehand that the achieved results follow a non-normal distribution by applying the Shapiro-Wilkinson test [26]. **We answer RQ2** by grouping the existing results by type of MT. On the MT-groupings we use the Friedman test [27] and the post-hoc Nemenyi test [28]. The Friedman test tests for significant differences among the different MTs in terms of their impact on the BLEU4-score. Furthermore, we use the post-hoc Nemenyi test

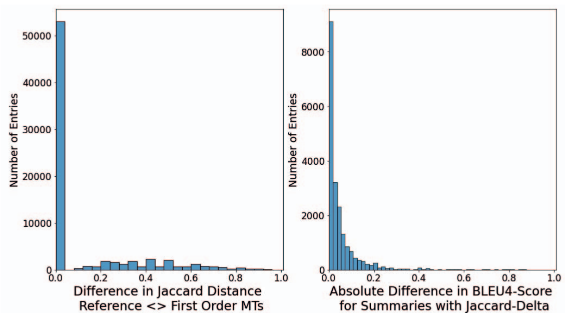

Figure 2. Overview of changes for first-order MTs

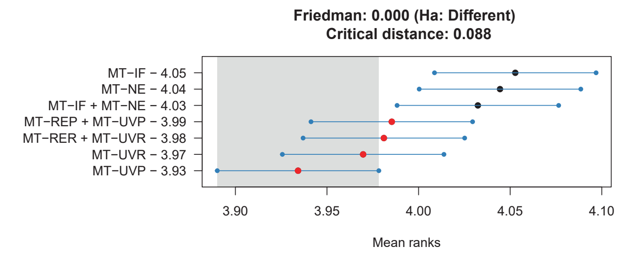

Figure 3. Results of the Friedman test and Nemenyi post-hoc procedure for different MTs.

to perform a pairwise comparison. The Nemenyi test measures the difference across the MTs by computing the average rank of each treatment across all datapoints in the test set, where the lowest rank is the most significant.

V. Preliminary Results

Results for RQ1. Figure 2 shows the histogram of deltas in the BLEU4-Score produced by CodeBERT before and after applying MTs. The Figure also shows the histogram of Jaccard distance between the reference and the post-MT generated JavaDoc. We observe that out of 72,989 produced JavaDoc summaries, for 16,566 datapoints CodeBERT generated summaries with a non-zero delta in BLEU-Score (22.6%). For these 16,566 datapoints with changes, the average difference in BLEU-Score is 0.06. Many summaries change when we apply the MTs, but they perform comparably in terms of BLEU-Scores to the unaltered; For example, both summaries could miss the same number of keywords, just by different tokens. More in detail, there are 52,838 Java methods in the test set out of 72,989 with zero Jaccard distance. This results in 20,151 snippets that do not pass the metamorphic tests (27.6%). Finally, the Wilcoxon rank-sum test revealed that there is a statistically significant (p-value$<$0.01) difference in the BLEU-Scores achieved by CodeBERT for the code snippets with changes between pre- and post-transformations.

Results for RQ2. We applied the Friedman test and the post-hoc Nemenyi procedure to analyze the impact of the different MTs on the BLEU-Score. With a p-value$<$0.01, the Friedman test indicates a statistical difference across the different types of MTs. The results of the post-hoc Nemenyi test are reported in Figure 3. From the ranking, we can see that the most impactful transformations are MT-UVR and MT-UVP, while MT-IF and MT-NE are the least impactful on the BLEU-Score. In terms of significance, we can conclude that MT-UVP is statistically more impactful than the other MTs.

VI. Discussion

We presented an effective approach for testing the robustness of a model towards metamorphic transformations on source code. According to the empirical results, our approach is capable of producing significant changes in the summaries generated by CodeBERT, highlighting potential weaknesses in the model as it does not satisfy metamorphic relations. In other words, slightly different variants of the same program can lead to significantly different results. While in this paper we focus on the Code-To-Text tasks of CodeBERT, we expect the found implications to hold true for other down-stream tasks as well. This can be considered a call-to-arms for researchers and practitioners to test machine learning models trained on source code using metamorphic testing in addition to the traditional performance metric (e.g., accuracy). We envision that a handful robustness-criteria are defined for the next generation of ML-based program analysis solutions, documented and tested using metamorphic transformations. We encourage reviewers of future research to perform sanity checks on newly published models and add robustness as a mandatory attribute of being *SOTA*. Lampion can also be used to increase the size of the test-set by generating new program variants, without requiring human labeling. This could be potentially beneficial for SE tasks where labeling data is very expensive.

VII. Conclusion

This paper introduces metamorphic relations to test ML-models program analysis solutions. Using this technology, our objective is to gain further information on the model's behavior apart from the performance metric (e.g., accuracy). To achieve this, we presented a generic approach (Lampion) and applied it in a case study on CodeBERT's Code-To-Text tasks. To evaluate the case study, we perform various statistical tests to prove or disprove changes in the resulting performance metric.

Our approach and framework can empower experts and laymen alike to assess the robustness of their models and provide additional tests on quality. We tried to keep the approach ① lightweight in concept,② expendable in implementation (due to plug-in MTs),
③ independent of the task (any language and quality metric).

While our initial implementation is in Java, we expect that a re-implementation for any language is an easy task and the statistical analysis can be reused for most experiments.

VIII. Online Resources

The code for a sample metamorphic transformer, the grid experiment and the evaluation can be found on Github under the Lampion repository[1]. The model,cleaned test-set and post-transformation datasets can be found on SurfDrive[2].

[1]https://github.com/ciselab/Lampion
[2]https://surfdrive.surf.nl/files/index.php/f/8713322177

References

[1] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[3] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[4] S. Shafiq, A. Mashkoor, C. Mayr-Dorn, and A. Egyed, "Machine learning for software engineering: A systematic mapping," *arXiv preprint arXiv:2005.13299*, 2020.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[7] B. Li, M. Yan, X. Xia, X. Hu, G. Li, and D. Lo, *DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1571–1575. [Online]. Available: https://doi.org/10.1145/3368089.3417926

[8] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[10] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, pp. 867–872, 2008.

[11] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544 – 558, 2011, the Ninth International Conference on Quality Software. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003213

[12] M. Sharif, S. Mohsin, M. Y. Javed, and M. A. Ali, "Single image face recognition using laplacian of gaussian and discrete cosine transforms." *Int. Arab J. Inf. Technol.*, vol. 9, no. 6, pp. 562–570, 2012.

[13] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," *arXiv preprint arXiv:2103.11626*, 2021.

[14] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[15] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[16] T.-H. Jung, "Commitbert: Commit message generation using pre-trained programming language model," *arXiv preprint arXiv:2105.14242*, 2021.

[17] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[18] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[19] C. N. Vasconcelos, A. Paes, and A. Montenegro, "Towards deep learning invariant pedestrian detection by data enrichment," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2016, pp. 837–841.

[20] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec," *Proceedings of the 17th International Conference on Mining Software Repositories*, Jun 2020. [Online]. Available: http://dx.doi.org/10.1145/3379597.3387445

[21] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *arXiv preprint arXiv:1910.07517*, 2019.

[22] "Codexglue: A benchmark dataset and open challenge for code intelligence," 2020.

[23] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[25] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[26] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[27] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, vol. 32, no. 200, pp. 675–701, 1937.

[28] P. Nemenyi, "Distribution-free mulitple comparisons phd thesis princeton university princeton," 1963.