Test Oracle Automation in the era of LLMs

Facundo Molina
IMDEA Software Institute
Madrid, Spain

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain

ABSTRACT

The effectiveness of a test suite in detecting faults highly depends on the correctness and completeness of its test oracles. Large Language Models (LLMs) have already demonstrated remarkable proficiency in tackling diverse software testing tasks, such as automated test generation and program repair. This paper aims to enable discussions on the potential of using LLMs for test oracle automation, along with the challenges that may emerge during the generation of various types of oracles. Additionally, our aim is to initiate discussions on the primary threats that SE researchers must consider when employing LLMs for oracle automation, encompassing concerns regarding oracle deficiencies and data leakages.

ACM Reference Format:
Facundo Molina and Alessandra Gorla. 2024. Test Oracle Automation in the era of LLMs. In *Proceedings of International Workshop on Software Engineering in 2030 (SE 2030).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The goal of software testing is to find defects. To actually find defects, though, test suites require relevant test inputs, i.e., inputs that can exercise the software under test (SUT) in realistic scenarios. Furthermore, to ensure that the SUT exhibits the expected behavior for these inputs, accurate test oracles are required. The problem of automating the generation of test oracles, the so-called oracle problem [4], has become very relevant in the last decades, as it has the potential to improve the oracles used in the testing process, and therefore contributing to revealing defects in software.

Various approaches have been proposed to address the oracle problem by automatically deriving different kinds of oracles [1, 3, 5, 9, 10, 22, 23, 35, 40], including *test assertions*, *contracts* (such as pre/postconditions and invariants) or *metamorphic relations*. Generally, these approaches observe some artifact related to the SUT (documentation, comments, source code, executions) and then derive oracles that are consistent with the observations. For example, TOGA [9] observes the source code of a target test and a focal method (method under test) and infers a test assertion for the given test; MeMo [5] extracts metamorphic relations by observing natural language comments in the source code; Daikon [10] and related tools [22, 23, 35] observe the behavior of the SUT (from a set of tests) in order to infer class invariants and pre/postconditions.

SE 2030, November 2024, Puerto Galinàs (Brazil)

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Despite all these efforts, the problem of automatically deriving oracles is still an open research problem in software testing [4]. The main reason is that automatically derived oracles are rarely accurate. These accuracy issues can result in high false positive rates, which can lead to false alarms and reduce the trustworthiness of the testing process. Indeed, recent studies have shown that even state-of-the-art neural based approaches can produce oracles with high false positive rates [13].

Large Language Models (LLMs) have already been used in tackling diverse software testing [38], demonstrating remarkable proficiency, particularly in tasks such as automated program repair [11, 16, 41] and automated test generation [18, 27, 29]. Given the positive results, researchers have also started to explore the use of LLMs to automatically generate test oracles, mainly in the form of test assertions [24, 25, 37]. Although the initial results are promising, showing that generated assertions can improve test coverage [37] and some lexical/functional metrics [25], there are still aspects of LLM-generated oracles that need to be further explored.

This paper aims to enable discussions on the use of LLMs for test oracle automation in general (not only test assertions) by discussing:

- the potential of LLMs for test oracle automation (including oracles that go beyond test assertions, such as contracts or metamorphic relations) and the challenges when using the LLMs through prompt engineering or by pre-training or fine-tuning them; and
- the main threats that arise from the use of LLMs to generate different kinds of oracles, including oracle deficiencies and privacy-related issues related to data leakages, and how we can mitigate such threats.

2 LLMS FOR ORACLE AUTOMATION

The most straightforward application of LLMs for oracle automation involves prompt engineering, wherein a prompt is designed to instruct any state-of-the-art pre-trained model, such as ChatGPT-3.5 [26] or Llama2 [36], to produce an oracle. The majority of studies in the literature that utilize LLMs for software testing tasks, as well as for other applications in general, concentrate on two primary strategies for prompt engineering: zero-shot learning and few-shot learning [38]. Zero-shot learning consists of providing a prompt asking the model for results, e.g., "Generate assertions for the following test: ...". Few-shot learning, instead, consists of providing a set of high-quality examples to the model. For instance, one could provide a set of test-assertion pairs for the model to generate assertions for a given test.

A more sophisticated approach to oracle automation via LLMs involves **pre-training or fine-tuning**. Pre-training entails training the model on a broad distribution of data to predict the subsequent token in a sequence. Conversely, during fine-tuning, the weights of a pre-trained model are adjusted by retraining it on a designated dataset tailored for a specific task. A clear example of this approach

has been proposed by Tufano et al. [37], where an LLM is pre-trained with a large source code and English language corpora, and then fine-tuned for generating assert statements.

Regardless of the strategy used, generating oracle via LLMs requires the use of data related to the expected oracles, either to build prompts or to pre-train or fine-tune the model. Various sources of pertinent information (such as source code, test code, documentation, logs, etc.) may be utilized to feed the LLMs. However, the type of information employed will also vary based on the type of oracle intended to be generated, thereby presenting distinct challenges. For instance, employing a zero-shot learning approach to produce a test assertion for a given test merely requires furnishing a prompt containing the test case and instructing the model to expand it with assertions. Applying the same methodology to generate broader oracles, such as postconditions for a method, necessitates a more detailed prompt. In addition to incorporating the method code, one must also specify the formalism or language for articulating the contract (e.g., as a code fragment, an assert statement, a logical expression, etc.).

To properly illustrate these challenges, we now consider the most common types of automatically inferred oracles: test assertions, contracts, and metamorphic relations.

2.1 Test Assertions

Test assertions in unit tests check the expected behavior of the SUT in a specific scenario, and are typically expressed as code. Given their importance for having high quality test suites, the automated generation of test assertions has been widely studied, including approaches that pre-train and fine-tune the models [25, 37]. Tufano et al. [37] pre-trained a BART Transformer model [19] with a large corpus of English text and Java code, and then fine-tuned it on the task of generating assert statements for unit test cases. Similarly, TeCo [25] fine-tunes the CodeT5 [39] and CodeGPT [20] LLMs specifically for the test completion task (i.e., predict the next statement in a test case), which are then evaluated for assert statement generation from the code under test (including the method under test), the test method signature, and the prior statements before the assertion statement.

These approaches can achieve an exact match rate (percentage of generated assertions that exactly match the expected assertions) of up to 62%. The recent improvements achieved by state-of-the-art LLMs enable the use of such models for generating test assertions using a more straightforward approach, such as zero-shot learning. Consider for example the test in Figure 1, which shows a very simple test case for a `Stack` class, with a unique test assertion checking that the stack is not empty after three push operations and one pop operation.

```
public void testPop() {
  Stack<Integer> stack = new Stack<>();
  stack.push(2);
  stack.push(3);
  stack.push(5);
  stack.pop();
  assertFalse(stack.isEmpty());
}
```

Figure 1: A simple test for a `Stack` class.

If we provide the prompt "Extend the following Java test just with assert statements: + test-code" where test-code is the test `testPop`, ChatGPT-3.5 produces the following assertions:

```
assert !stack.isEmpty() : "Stack should not be empty after pop";
assert poppedElement == 5 : "Popped element should be 5";
```

also checking that the popped element is 5 (ChatGPT-3.5 edited the test to save the result of pop in the variable `poppedElement`). In fact, Nashid et al [24] recently proposed a technique for prompt creation based on embedding or frequency analysis, that can achieve an exact match rate of 76% in test assertion generation.

2.2 Contracts

Contracts [21] are logical constraints on a specific software element (method, class, etc.), and are usually captured in the form of preconditions and postconditions for methods, or representation invariants for classes. As opposed to test assertions, contracts are not specific to a particular test case, but rather express properties that must hold for any execution. Moreover, contracts can be expressed in different formalisms, not only in the same language as the SUT. For instance, JML [6] is a popular library for Java that includes a language for writing contracts as annotations in the source code. Daikon [10], a well-known tool for dynamic invariant detection, produces pre/postconditions and class invariants using its own language (a mix of Java, and mathematical logic). Other contract inference tools also use their own languages [22, 23, 35].

To the best of our knowledge, there are still no studies using LLMs to generate contracts. To give an idea on how LLMs could be used for this task, let us consider an implementation of the `push` operation for a `Stack` class, shown in Figure 2.

```
public void push(E e) {
  if (size == elements.length) {
    ensureCapacity();
  }
  elements[size++] = e;
}
```

Figure 2: Implementation of a `push` operation for a `Stack`.

The code first ensures that there is enough capacity in the `elements` array, and then pushes the element in the next available position. A contract for this method could include a postcondition stating that the size of the stack is increased by one, and that the element is added at the top (`size-1`) of the stack.

The variety of formalisms for expressing contracts poses a challenge when using LLMs, mainly because one would also need to specify the formalism in which the contract must be expressed. State-of-the-art pre-trained models have the potential of producing contracts with a reasonable performance, using a prompt engineering approach, at least for well-established formalisms. For instance, with the zero-shot prompt "Generate a postcondition in the form of an assert statement for the following method: + method-code", where method-code is the `push` method in Figure 2, ChatGPT-3.5 produces the following postcondition:

```
assert elements[size - 1].equals(e) : "Element was not
     successfully pushed onto the stack.";
```

Note that although the postcondition captures one expected property, its execution will crash if the pushed element is `null`. Similarly, if we ask for the postcondition to be expressed in JML, the model produces the same postcondition.

However, for less known formalisms or subjects with a more complex API, using a few-shot prompt, or even pre-training and fine-tuning the model, could be more appropriate. While using a few-shot prompt would require one to provide a set of examples of the expected contracts (e.g., a set of methods with their contracts expressed in the desired formalism), the use of pre-training or fine-tuning would require one to provide a specific and extensive dataset for contract inference, including the software element (method or class), the expected contract, and possibly other contextual information related to the contract to learn.

2.3 Metamorphic Relations

Metamorphic relations express domain-specific properties of multiple executions of the SUT [30]. Compared to test assertions and contracts, they are more general oracles that are easy to define and mantain. Some examples of metamorphic relations involving two executions are $p(x, y) = p(y, x)$ for a commutative operation p, or $sort(a) = sort(b)$ for a sorting operation where a is a permutation of b. Since its introduction by Chen et al. [8], metamorphic relations have been successfully used for detecting bugs in a variety of software systems, including the search engines Google and Bing [42] and the Web APIs of Spotify and YouTube [31]. Moreover, metamorphic relations have shown to be complementary to other types of oracles, such as test assertions, to detect faults in the SUT [5].

According to the literature [30], metamorphic relations are expressed through some formalism that allows to capture the expected relationship between the inputs and outputs. Metamorphic relations are typically instantiated in a test case, where a *source test* (e.g., $a1 = sort([1, 3, 2])$) is executed, then a *follow-up test* (e.g., $a2 = sort([2, 1, 3])$) is executed, and finally the relation is checked ($a1 = a2$) [32].

For the inference of metamorphic oracles, the use of LLMs has not yet been explored. Using LLMs to generate metamorphic relations can be challenging, mainly because of the domain-specific knowledge required to identify the relations, and the lack of well-defined formalisms to express them. A straightforward approach could be enabled in the case that the SUT already contains a set of test cases, as these could be used as source tests to ask an LLM to generate follow-up tests that preserve some relation with respect to them. For example, using the zero-shot prompt "Generate a follow-up test that is equivalent to the following test + test-code" with the `testPop` test from Figure 1, ChatGPT-3.5 produces a test with exactly the same operations, and two additional sentences:

```
stack.push(7);
stack.pop();
```

The produced follow-up test is equivalent to the source test in the sense that both result in the same stack. It is easy to see that the metamorphic relation behind states that for every stack, if we push an element and then pop it, the stack remains the same. Using both tests, one could easily implement a new test to check the metamorphic relation by first executing the source test, then the follow-up test, and finally checking that the stacks are equal.

If a set of test cases is not available, or one wants to focus on generating the metamorphic relation itself (instead of its implementation), one would need to include in the communication with the LLM how the relations should be expressed. For this scenario, a few-shot prompt or a pre-training and fine-tuning approach would be necessary. Independently of the kind of oracle we are generating, it is evident that LLMs have an enormous potential to assist in oracle automation. However, as we discuss in the next section, the use of LLMs for this task not only inherits some threats from previous techniques that can affect the quality of the produced oracles, but also introduce new threats that need to be considered.

3 THREATS TO VALIDITY

3.1 Oracle Deficiencies

The quality of an oracle can be assessed in terms of its oracle deficiencies: false positives and false negatives [15]. Intuitively, a **false positive** is a correct and expected program state for which the oracle is false, i.e., a false alarm. A **false negative** is an incorrect and unexpected program state for which the oracle is true, i.e., a missed fault. While false negatives are more tolerable, as one can still fix the oracle to improve its fault detection capability, false positives are more critical, as they can lead to false alarms resulting in unnecessary debugging efforts.

Failing to identify and eliminate false positives may result in high false positive rates, and raise concerns about the practical usefulness of the tools [13]. To mitigate this, we consider that assurance mechanisms post-processing LLM-produced oracles are imperative to provide guarantees on their quality, and may be crucial to minimize oracle deficiencies. Alshahwan et al. [2] recently proposed the notion of Assured LLM-based Software Engeering (Assured LLMSE) with the aim of providing guarantees on the output produced by LLMs used for software engineering tasks. Figure 3 shows an overview of how Assured LLMSE could be applied to LLM-based oracle generation, where the LLM-produced oracles are subjected to an assurance process, possibly analyzing deficiencies. The assurance process can also provide information to re-prompt the model with those oracles that do not pass the process.

As LLMs offer absolutely no guarantees on their outcome, the assurance process could first easily eliminate syntactically incorrect oracles. Detecting oracle deficiencies, however, is more challenging. False positives detection depends on the usage context. If the oracles are for regression testing, i.e., to equip the current version of the SUT with oracles in order to detect regression errors in the future, to detect false positives one can search for a reachable program state (test case) in which the oracle fails. For example, GAssert [35] uses evolutionary computation to actively search for test cases that falsify a given assertion oracle. On the other hand, if the oracles are intended to test the current implementation, the detection of false positives may require a human to decide whether an oracle failure indicates a false positive or a real bug.

Once false positives are detected, one can fix the oracle either by removing the parts that are incorrect, or by weakening the oracle to make it more general. This could enable an iterative process, possibly re-prompting the LLM, in which the oracle is refined until no more false positives are detected. This iterative refinement process has already been applied for postcondition assertions [35].

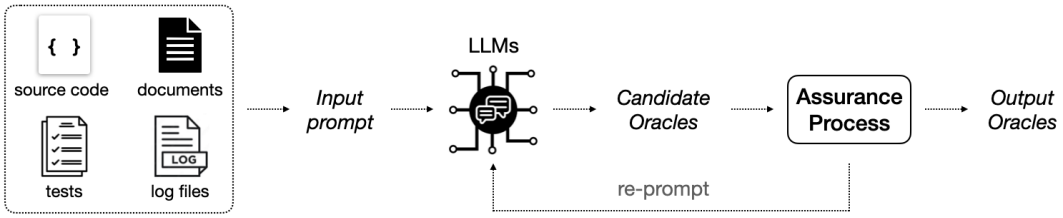

Figure 3: Overview of LLM-based Oracle Generation.

Although false negatives are less critical, their detection can help to strengthen the oracle, improving its fault detection capability. As false negatives are essentially faulty program states missed by the oracle, mutation analysis can be used to introduce artificial faults, and then check if the oracle can detect them. While some approaches use mutation analysis to improve the oracles during the inference process [12, 22, 35], others use it as an evaluation technique for fault detection analysis [1, 5]. Similarly, LLM-based techniques could also benefit from false negatives analysis.

3.2 Oracle Leakages

The use of LLMs for software engineering tasks has several threats, including closed-source models, data leakages between training data and research evaluation, and reproducibility issues [28]. Among these, data leakages is particularly relevant for LLM-based oracle automation, specifically during the evaluation of these approaches.

As LLMs are trained on very large amounts of data, often including publicly available code from GitHub [7], there is a risk that the model has memorized some code samples from the training data [14]. Thus, when evaluating LLM-based techniques for oracle automation, we need to pay special attention to the data we use for evaluation, as we may end up obtaining oracles that are not actually created by the model, but rather replicated from the training data.

To illustrate this issue, let us consider Defects4J [17], one the most widely-adopted benchmarks in software testing research. Many of the projects involved in Defects4J are publicly available on GitHub. Thus, evaluating LLM-based oracle generation techniques on Defects4J can clearly lead to oracle leakages, and make the LLM provide accurate oracles just because they are a copy of the oracles from the training data. Figure 4 shows an example of a test prefix (i.e., the first part of a test without the assertions) from the Apache Commons Collections project in Defects4J, available in revision `7c99c62` of the project repository[1], and the test assertions generated by ChatGPT-3.5. With the prompt "Complete the following Java test with test assertions: + test-code", ChatGPT-3.5 produces exactly the same assertions as in the original test case, with the sole difference that natural language messages to explain the expected behavior are included. Moreover, the model uses the `search` method from the `ArrayStack` class in the assertions, which availability was not even informed in the initial prompt.

To mitigate oracle leakages in the evaluation of LLM-based techniques for oracle automation, one could consider using evaluation data from multiple sources, as recommended by Sallou et al. [28].

[1]https://github.com/apache/commons-collections

```
public void testSearch() {
  final ArrayStack<E> stack = makeObject();
  stack.push((E) "First Item");
  stack.push((E) "Second Item");
-----------------------------------------------------------------------
  // Test searching for existing elements
  assertEquals(2, stack.search("First Item")); // First Item is
     at index 2 from the top
  assertEquals(1, stack.search("Second Item")); // Second Item
     is at index 1 from the top

  // Test searching for non-existing element
  assertEquals(-1, stack.search("Non-existing Item")); //
     Non-existing Item is not found in the stack
```

Figure 4: Test from Defects4J, and the assertions produced by ChatGPT-3.5.

SourceForge projects are potentially a good source of data for evaluation, as they have been shown that LLMs can have a worse performance on them compared to GitHub projects [33]. Moreover, one could also consider the use of datasets containing code produced after the models training, to ensure that the evaluation code has not been seen by the model. For example, GitBug-Java [34] is a recent benchmark of recent Java bugs built with 2023 code, which is after the cut-off date of the training data of most of the notable LLMs, including OpenAI models.

4 CONCLUSION

Thanks to the ability of LLMs to quickly generate content, either as code or as specifically formatted text, they have an enormous potential to improve software testing tasks.In this paper, we discuss the potential of LLMs for test oracle automation, along with insights to deal with the challenges present in the use of the LLMs for inferring different types of oracles.

We also discuss the main threats that arise from automatically generating oracles using LLMs. Using the LLMs without an assurance process can result in low quality oracles, containing a high number of oracle deficiencies, mainly false positives, which can reduce the trustworthiness of the testing process. Moreover, new threats can emerge from the use of LLMs, such as oracle leakages, which can lead to the generation of oracles that are not actually created by the model, but are rather replicated from the training data.

We believe that an Assured LLM-based Software Engineering approach can be a promising direction to mitigate the threats of LLM-based oracle generation, and to provide guarantees on the quality of the oracles produced.

REFERENCES

[1] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1018–1030. https://doi.org/10.1145/3597926.3598114

[2] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering. *CoRR* abs/2402.04380 (2024). https://doi.org/10.48550/ARXIV.2402.04380 arXiv:2402.04380

[3] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485481

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/j.jss.2021.111041

[6] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 342–363. https://doi.org/10.1007/11804192_16

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[8] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. In *Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.*

[9] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2130–2141. https://doi.org/10.1145/3510003.3510141

[10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[11] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[12] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *ISSTA*. ACM, 147–158.

[13] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian G. Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 120–132. https://doi.org/10.1145/3611643.3616265

[14] Huseyin A. Inan, Osman Ramadan, Lukas Wutschitz, Daniel Jones, Victor Rühle, James Withers, and Robert Sim. 2021. Training Data Leakage Analysis in Language Models. (2021). arXiv:2101.05405 [cs.CR]

[15] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[16] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[17] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[18] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[19] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR* abs/1910.13461 (2019). arXiv:1910.13461 http://arxiv.org/abs/1910.13461

[20] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. (2021). arXiv:2102.04664 [cs.SE]

[21] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[22] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1008–1020. https://doi.org/10.1145/3510003.3510120

[23] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[24] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[25] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2111–2123. https://doi.org/10.1109/ICSE48619.2023.00178

[26] OpenAI. 2024. *OpenAI*. https://openai.com/ Accessed on March 13th, 2024.

[27] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 409–420. https://doi.org/10.1109/ASE56229.2023.00193

[28] June Sallou, Thomas Durieux, and Annibale Panichella. 2023. Breaking the Silence: the Threats of Using LLMs in Software Engineering. *CoRR* abs/2312.08055 (2023). https://doi.org/10.48550/ARXIV.2312.08055 arXiv:2312.08055

[29] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[30] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[31] Sergio Segura, José Antonio Parejo, Javier Troya, and Antonio Ruiz Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Trans. Software Eng.* 44, 11 (2018), 1083–1099. https://doi.org/10.1109/TSE.2017.2764464

[32] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Softw.* 37, 3 (2020), 46–53. https://doi.org/10.1109/MS.2018.2875968

[33] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. (2024). arXiv:2305.00418 [cs.SE]

[34] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. (2024). arXiv:2402.02961 [cs.SE]

[35] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. https://doi.org/10.1145/3368089.3409758

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov,

Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/arXiv.2307.09288 arXiv:2307.09288

[37] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 54–64. https://doi.org/10.1145/3524481.3527220

[38] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]

[39] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[40] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 191–200. https://doi.org/10.1145/1985793.1985820

[41] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[42] Zhiquan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Trans. Software Eng.* 42, 3 (2016), 264–284. https://doi.org/10.1109/TSE.2015.2478001