UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing

Yifeng He
University of California at Davis
Davis, USA
yfhe@ucdavis.edu

Jiabo Huang
Tencent
Shenzhen, China
jiabohuang@tencent.com

Yuyang Rong
University of California at Davis
Davis, USA
PeterRong96@gmail.com

Yiwen Guo
Unaffiliated
China
guoyiwen89@gmail.com

Ethan Wang
University of California at Davis
Davis, USA
ebwang@ucdavis.edu

Hao Chen
University of California at Davis
Davis, USA
chen@ucdavis.edu

Abstract

The remarkable capability of large language models (LLMs) in generating high-quality code has drawn increasing attention in the software testing community. However, existing code LLMs often demonstrate unsatisfactory capabilities in generating accurate, complete tests since they were trained on code snippets collected without differentiating between code for testing and for other purposes. In this paper, we present a large-scale dataset, UniTSyn, which can enhance LLMs for **Uni**t **T**est **Syn**thesis. Associating tests with the tested functions is crucial for LLMs to infer the expected behavior and the logic paths to be verified. By leveraging Language Server Protocol, UniTSyn achieves the challenging goal of collecting focal-test pairs without per-project execution setups or per-language heuristics, which tend to be fragile and difficult to scale. Containing 2.7 million focal-test pairs across five mainstream programming languages, it can enhance the test generation ability of LLMs. Our experiments demonstrate that, by building an autoregressive LLM based on UniTSyn, we can achieve significant benefits in learning and understanding unit test representations, resulting in improved generation accuracy and code coverage across all the evaluated programming languages.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Neural networks**.

Keywords

Large language models, software testing, test case generation, dataset

ACM Reference Format:
Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680342

1 Introduction

Software testing is a crucial yet labor-intensive part of the software development process [68, 4]. The importance on testing on early detection of program defects has been established for decades [13]. Recently, machine learning models, especially large language models (LLMs), have shown their prowess in composing high-quality code [34, 55], which has further fueled the software testing community's interest [56, 50] in applying LLMs to unit test generation or other program testing applications.

Test generation is more challenging than conventional programming synthesis, as the model needs to generate not only executable code snippets but also precise predictions of input-output values. Equipping off-the-shelf code LLMs [61, 69, 44] with prompts or instructions for test generation is poorly suited, as demonstrated in previous work [50]. A few recent projects [43, 50] targeted training testing-specific LLMs on code corpus that is closely related to testing. A desired training set for this aim should include a large number of test functions paired with their tested source functions, also called *focal functions* [59, 56].

However, there are significant challenges surrounding the automation of collecting focal-test pairs since real-world projects do not have to follow a consistent structure. Existing efforts either rely on dynamic analysis or heuristics to locate focal functions [14, 43, 59] or find the coarse-grained correspondence between tests and focal functions at the file level [50]. The former category is *difficult to scale* across programming languages, which inhibits the development of universal testing models. On the other hand, the latter category is limited by *weak focal-test correspondences*, which hamper the models' capability to properly comprehend the expected behavior and logic paths being verified. These challenges underscore the need for more effective, scalable, language-agnostic approaches to collect pairwise focal-test data to fully unleash the potential of LLMs on software testing.

We present UniTSyn as a multilingual dataset capable of enhancing LLMs for **Uni**t **T**est **Syn**thesis. As shown in Figure 1, we integrated the Language Server Protocol (LSP) into the dataset-building process to harness its language extensibility and call-definition

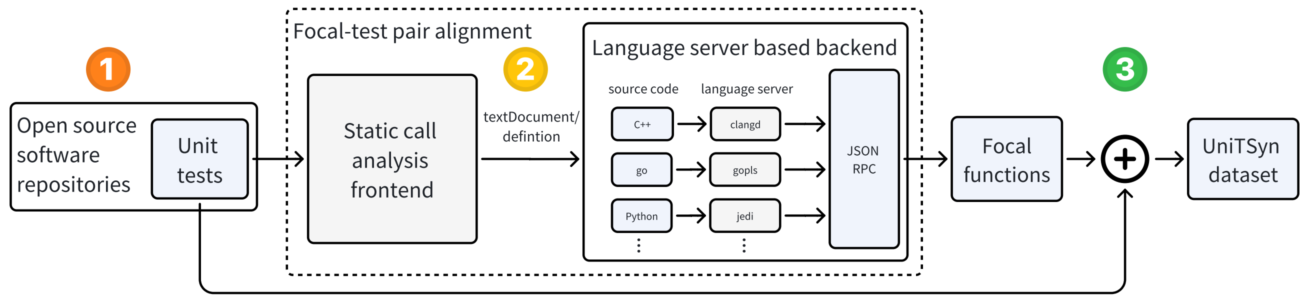


Figure 1: UniTSyn overview.
1. We download open-source software repositories and extract their unit tests.
2. We use static analysis to identify the call to their focal functions and use the language server protocol to get the location of the focal function definition.
3. We store the aligned function-level focal-test pairs as training data.

Table 1: Dataset statistics.
Framework: static analysis for test extraction
#Proj: number of projects found on GitHub for each language
#Pairs: number of focal-test pairs collected for each language

Language	Framework	#Proj	#Pairs
Python	unittest, pytest	43 848	1 218 311
Java	JUnit	25 488	1 097 518
Go	testing	38 097	361 075
C++	GoogleTest	20 090	25 513
JavaScript	MochaJS	17 621	13 293

Table 2: Datasets comparison.
#Proj: number of software projects in the dataset
#Lang: number of programming languages in the dataset
Unit Test: if the dataset specifically mines testing code
Alignment: the level of alignment between testing code (if exists) and code to be tested.

	The Stack [30]	CAT-LM [43]	TeCo [50]	UniTSyn (ours)
#Proj	137.36M	197 730	1270	246 194
#Lang	30	2	1	5+
Unit Test	✗	✓	✓	✓
Alignment	✗	file	function	function

matching ability. This substantially eases the difficulty of implementing dependency analysis heuristics for different languages and executing different projects for dynamic analysis. Moreover, we designed a flexible, unified static analyzer to find calls to focal functions from the unit tests, which decreases the complexity of performing call analysis for each language.

To explore the quality of the UniTSyn dataset, we further trained an autoregressive model called UniTester to synthesize tests in different programming languages. UniTSyn yielded significant performance advantages on generating accurate, complete tests over several state-of-the-art LLMs intended for both code and test synthesis, which demonstrates that UniTSyn is a flexible dataset collection framework.

To sum up, we make the following contributions:

(1) We provided a large-scale dataset of 2.7 million focal-test function pairs across five commonly used programming languages, which will be useful in advancing the field of software engineering through LLM coding assistance.
(2) We released our generic and easily applicable approach for building multilingual unit test datasets with function-level focal-test alignment. Our approach is extendable to any language that has a mature implementation of LSP, allowing the LLM to broaden its testing capabilities in more diverse software engineering scenarios.
(3) We validated the quality of UniTSyn by training an autoregressive language model on it. The model generated more accurate, complete tests compared with existing test and code LLMs, which demonstrates the necessity and benefits of training with explicit correspondence between tests and focal functions for multilingual test generation.

2 Related Work

2.1 Code Understanding and Generation

The application of Machine Learning (ML) in Software Engineering (SE) has gained significant attention recently, particularly with the development of LLMs. LLMs can help SE in various ways, including code generation [7, 35, 45, 34, 55, 62], code summarization [25, 33, 1], and code classification [15, 20, 67, 23]. To facilitate the study of ML for SE, a variety of datasets have been built. These datasets are either collected from competitive programming contests, like POJ104 in the CodeXGlue benchmark [38] and CodeNet [48], or open-source software like the CodeSearchNet challenge [24], CoderEval [66], and The Stack [30]. Datasets specialized for evaluating the code generation performance of LLMs like HumanEval [7] and HumanEval-X [69] have also been established using algorithmic coding problems formatted just like on LeetCode. These datasets are built for general-purpose program synthesis like CodeT5+ [61], CodeGen2

[44], InCoder [17], and SantaCoder [2]. However, as shown by [67] and [23], test cases can greatly help improve the model's ability to understand code. With few datasets focusing on test cases, the necessity of UniTSyn is justified.

2.2 Software Testing

2.2.1 Unit Testing. Unit testing is a common self-assessment testing technique where developers use a set of inputs and outputs of their code to validate that the code is working as expected [71]. In this setting, a test case consists of an input and the corresponding output after the code execution. The classical style of unit testing includes three phases: arrange, act, and assert [27]. This design pattern makes the important action (invoking the focal function) comes after arrangements of values and right before assertion. Therefore, well-crafted unit testing functions usually have the call to the focal function before the assertions. To evaluate the completeness and comprehensiveness of test cases, code coverage is often used as a common metric [65, 42]. Code coverage measures the percentage of the code that is executed. Statement, line, and branch coverage are often used depending on the coarseness of the testing requirements. Code coverage is measured because executing a piece of code is the necessary condition for finding bugs in it [42]. In SE, coverage-guided software testing has also shown its power in detecting bugs in various software domains [10, 9, 57, 8, 16, 53, 54, 36], highlighting the importance of this metric. In general, unit testing is an indispensable part of the modern software development ecosystem, ensuring the quality and reliability of systems across various domains. Therefore, generating test cases with high coverage is a necessity. As demonstrated in our experiments, our dataset is capable of enhancing the ability of LLMs to generate such test cases.

2.2.2 Property-based Testing. Property-based testing is a hybrid approach towards self-assessment testing and randomized testing, where the developers identify a set of properties for the program to satisfy and the framework generates suitable inputs randomly. Instead of writing individual test cases or examples that check for fixed input-out pair as unit tests, developers write tests that verify if certain properties hold for a wide range of randomly generated inputs. The idea of property-based testing was first proposed in QuickCheck [11] for testing Haskell. Although it was designed for functional programming with monadic abstraction, property-based testing was later adopted by other languages in UniTSyn [40, 26, 18, 51, 47]. Due to its difference in style compared to unit tests, we have not made any special treatment for property-based testing functions, yet the collected dataset may also cover some code for property-based testing and thus may benefit it as well, which can be testified in detail in future work.

2.3 Software Testing via Machine Learning

The goal of test generation is to utilize ML models to aid software testing, which can be achieved via prompting or instructing general-purpose code LLMs [61, 69, 44], or training testing-specific LLMs. ATLAS [63], AthenaTest [59], TOGA [14], TeCo [43], and CAT-LM [50] are testing-specific LLMs based on the transformer architecture. Some testing-specific LLMs are trained on large-scale test functions and their aligned focal functions [43, 50], where focal functions are the functions being tested [59, 56]. To align test and focal functions, some work relies on dynamic execution context or heuristics for locating focal functions [14, 43, 59]. The extensibility of this approach is limited, since automating the setup of dynamic execution for different projects is challenging even within the same language. For example, TeCo's execution-based data collection was only applied to 1270 Java projects, which is one of the easiest languages for cross-platform execution thanks to the Java Virtual Machine and the Maven build system. The build system sets up the project in a way that makes it easy to automate the execution. Extending this method to other languages would likely be prone to complications not present in TeCo. On the other hand, relaxing the alignment at file-level [50] is easier to scale up than the previous approach. However, this weak focal-test correspondence disrupts the models' ability to thoroughly understand the expected behavior and logic paths in the focal function. We summarize the characteristics of these popular datasets for code language models and test generations in Table 2.

3 Design of UniTSyn Dataset

3.1 Challenges

As the interest in using LLMs for test generation grows, the inherent limitations of their underlying datasets are becoming increasingly apparent. Models like CAT-LM [50], specifically designed for test generation, suffer from a lack of flexibility to adapt to various languages due to dataset constraints. By contrast, LLMs trained on general-purpose code datasets like SantaCoder [2] can incorporate new languages by pulling code from platforms like GitHub. However, their training data often lacks a crucial link between the test functions and the focal functions. Without this correspondence, the models face challenges in deducing the intended behavior and logic paths of focal functions when generating tests. Consequently, this leads to tests that are not precise or thorough. we aim to build a dataset that emphasizes unit tests and their corresponding focal functions while being multilingual. This leads to two major challenges. First, analyzing tests for their focal function calls requires domain knowledge and language-specific rules since different languages have distinct grammars and unique syntax for unit tests. Second, extracting precise focal function definitions from the dependency graph is labor-intensive and unique to each language. To overcome these challenges, we abstracted the differences in languages away from the static analysis pipeline to build a generalized static call analyzer that operates on the Abstract Syntax Tree (AST) of different languages, and integrate the existing language servers to locate the focal function definition via their dependency analysis. This design not only reduces the difficulty of analyzing calls across different programming languages but also eliminates the requirement of implementing several dependency analyses or setting up multiple execution environments.

3.2 Data Collection

Our dataset contains a large assemblage of data collected from open-source software. We used CAT-LM's [50] approach to locate repositories on GitHub that are under active development. More specifically, we mined repositories in Python, Java, Go, C++, and JavaScript that have more than 10 stars, new commits after Jan 1st,

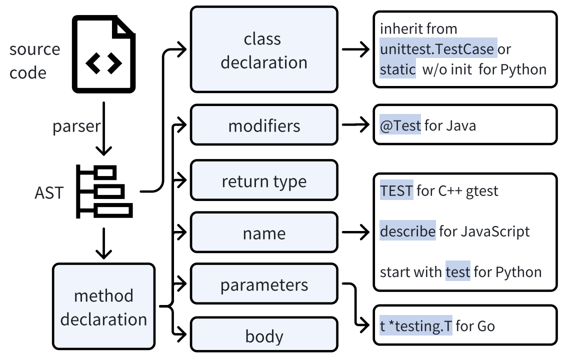


Figure 2: Test collection phase of the UniTSyn frontend. The highlighted words are identifiers for locating testing functions.

2020, and appropriate open-source licenses. to extend CodeSearchNet [24]. In addition, we filtered out repositories that are archived, forked, or mirrored from other repositories.

3.3 Dataset Construction

Providing more fine-grained data with multilingual focal-test alignment is useful to machine learning for understanding the implementation of focal functions [14, 43, 59]. We designed UniTSyn to achieve this goal. The frontend identifies potential test functions and locates the focal functions. The backend will then retrieve the source code of the focal function from the repository codebase. Figure 1 shows an overview of the workflow for obtaining our UniTSyn dataset. Our dataset construction method ensured that all the functions in the dataset were associated with at least one unit test function. This implies that the code in our dataset was the most important portion of their repositories and was checked by their developers intentionally

3.3.1 Parsing. To construct a multilingual unit test dataset, we need to generalize the static analysis process among different languages. The first step in any static analysis is to parse the source code into AST, which can be achieved using each language's compiler or interpreter. To ease the differences in invoking different tools for different languages, we selected tree-sitter [58] as the backbone of our parsing process. Tree-sitter can parse any programming language that has a formal syntax definition. Using the tree-sitter, we designed an easily extendable parsing and AST interface to enable smooth static analysis for all languages.

3.3.2 Test Function Identification. We traverse ASTs to find the test functions. To mitigate the differences between languages, we provide an interface to determine whether an AST contains any test functions. This interface takes hooks as call-back functions to check for test functions. In this work, we present two implementations. One relies on heuristics by checking the function name. The other uses language-specific features to determine if a function is a test. For example, in Java `JUnit`, all test functions need to have the `@Test` modifier. Figure 2 illustrates the test collection phase and the

```
1   @Test
2   public void testAdd() {
3       int x = 500;      int offsetX = 100;
4       int y = 700;      int offsetY = 200;
5       Position mp = new Position(x, y);
6       Position result = mp.add(offsetX, offsetY);
7       assertEquals(mp.x + offsetX, result.x);
8   }
9   public Position add(int x, int y) {
10      return new Position(this.x + x, this.y + y);
11  }
```

Listing 1: A Java test function and its paired focal

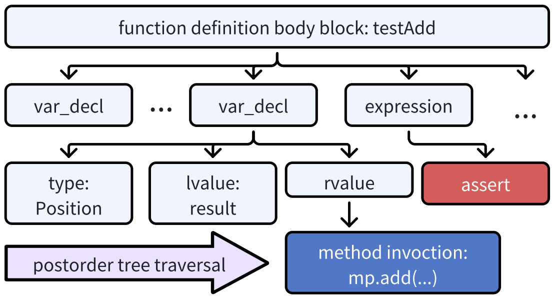


Figure 3: Focal function call analysis phase of UniTSyn frontend. This figure shows a simplified AST of the example test function in Listing 1. The red node in the AST is the first encountered assertion node with a postorder tree traversal. `var_decl` is the abbreviation of the variable declaration.

aforementioned hooks are summarized to the right of the figure. To extend this framework to other languages and testing suites or to improve the success rate of test function identification, one could provide a new, specially designed hook. Some projects may also include test functions that follow the practice of property-based testing [11] For example, Python test functions with `@given` decorator from the hypothesis [40] package. We treat these test functions the same as unit testing functions.

3.3.3 Focal Function Call Analysis. Given a test function, we need to identify its focal function call. Since a unit test function usually makes multiple function calls to set up its environment, it is hard to identify the real call to the focal function. In this paper, we followed TeCo's [43] practice to select the last function call that invokes a function definition in the repository before the first assertion. Listing 1 demonstrates an example of a Java test case, where line 6 includes the focal function call. To allow easy extension to different languages as a unified method, we designed another interface that the developers can implement with one extra function. The static call analysis phase to build UniTSyn is demonstrated in Figure 3. Our framework's generic focal-call analysis algorithm only requires one extra function to adapt to a new language. Our analysis algorithm employs a post-order tree traversal on the AST, starting from the root and progressing to the leftmost node. It visits all the subtrees before returning to the root. Consequently, our algorithm accurately identifies focal function calls within assertion statements, as the nodes representing these function calls are children of the assertion nodes.

3.3.4 Focal Function Extraction via LSP. We incorporated LSP (which is a language-neutral and standardized protocol for communication between language servers and editor clients [41]) into our dataset-building process to utilize its ability to decouple the features and functionalities of programming languages from integrated development environments [52] and its adaptability to different programming languages. This design helps relieve the pressure on dependency analysis across various languages and executing multiple projects. This separation allows the language intelligence to run as an independent service, and hence be integrated into our dataset construction. The `textDocument/definition` API in LSP resolves the definition location of a symbol at a given text document position. Therefore, we developed the UniTSyn backend as a client to the language servers, where we send `textDocument/definition` requests to the language servers to locate the definition of the focal function. The language servers send back the location of the focal function as a response [19]. Our approach of using LSP can easily extend to other languages as long as the language's language server is implemented. Our proposed method of aligning function-level focal-test pairs using LSP is easily extendable, requiring only the commands to start the corresponding language server.

3.4 Data Quality Analysis

We demonstrate the advancements in the quantity of focal-test pairs in our dataset, as detailed in Table 1. Additionally, the benefits of UniTSyn are demonstrated through its diversity in projects and languages, and its precise function-level focal-test alignment in Table 2. While large program datasets like The Stack [30] support many languages, they focus less on testing code and fail to offer clear guidance on the relationship between regular and test functions. The existing test-focused datasets either is *difficult to scale* across different programming language due to their design, or *align focal functions and tests weakly*. In this section, we further assess the quality of our data using empirical software engineering metrics.

3.4.1 Test-to-code Ratio. High-quality code repositories should be well-tested. We further analyzed the quality of our dataset by the test-to-code ratio of the projects we included. We compute and analyze the test-to-code ratio r as

$$r = \frac{LOC_{\text{test}}}{LOC_{\text{func}}}$$

where LOC_{test} is the lines of discoverable testing code and LOC_{func} is the lines of regular functional code. This ratio provides insight into the balance between testing code and the overall complexity of the software. We exclude projects with a test-to-code ratio of 0 when compiling the dataset. As illustrated in Figure 4, test codes significantly outnumber regular functional codes within our dataset. The distribution of the per-project test-to-code ratio for Python, Java, C++, and Go is mainly between 0 to 2. For JavaScript projects, the test-to-code ratio spreads more evenly. This broader distribution is due to the distinctive functional syntax of MochaJS, which structures a test function as a series of nested closures.

Additionally, the majority of projects in our dataset have a ratio $r \geq 1$, reflecting a wide diversity of test case implementations. This finding is also supported by our code coverage experiment with generated test cases in Section 4.3.2. Notably, the ratio distribution for Python projects is more skewed to the right compared to other languages. This skewness results from the behavior of the Python language server's `text/definition` request, which navigates to the entire class implementation rather than just the constructor as in other object-oriented languages. Such behavior results in a higher denominator for the ratio calculation, further affecting the distribution.

We also evaluated the ratio of focal functions paired with multiple test functions. In our dataset, 24% of focal functions are paired with more than one test function. If a focal method is paired with k different test functions, we treat them as k unique focal-test pairs during the continuous training stage of the model. Exposing the model to multiple test functions for the same focal function teaches it to generate more diverse test cases, achieving higher code coverage. This result is highlighted in Table 4.

3.4.2 Assertion Density. Adding assertions has a significant contribution in reducing defects in software [5]. Assertion density [32] is a common metric in measuring test code quality [3], and therefore is a good measurement to analyze the quality of our dataset. Assertion density d is calculated as

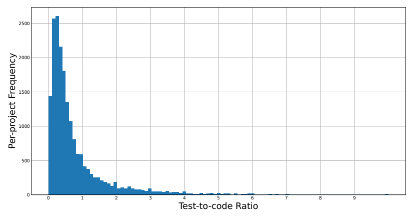
$$d = \frac{\#assertions}{LOC_{\text{test}}}$$

where $\#assertions$ is the number of assert statements in our dataset and LOC_{test} is the line of testing code in our dataset.

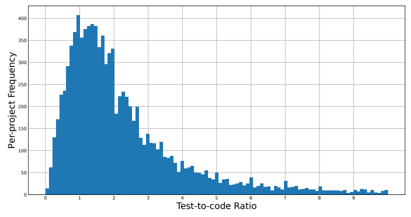
We analyzed the assertion density across various programming languages and summarized the findings in Figure 5. Our analysis indicates that test functions in Python and C++ exhibit notably higher assertion densities compared to other languages. The effect of this higher overall per-project assertion density is also shown in Section 4.3.4, where the Python subset demonstrates its contribution to model accuracy on the monolingual variant. The distribution of other languages centered around 0.2 (20%). In comparison to the statistics listed by Athanasiou, Nugroho, Visser, and Zaidman with a median of 8.4% and a mean of 9.1%, our dataset shows its quality with a higher assertion density in all languages.

3.4.3 Code Coverage of the Collected Test Functions. Code coverage is a crucial metric for assessing test adequacy [71, 27]. It thus also plays an important role in evaluating the quality of a unit test dataset. However, rebuilding open-source software (OSS) using a standardized command poses a significant challenge and remains a hot topic in software engineering research [21, 70]. This is due to each project requiring different versions of the compiler, interpreter, runtime, dependencies, etc. Furthermore, many OSS projects define unique procedures for building the software and executing tests.

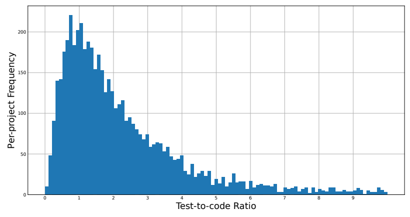
In our study, we sampled 10,000 Python projects to gather code coverage data. We established a consistent testing environment using a Docker container configured with Ubuntu 22.04 and Python 3.10. We executed the tests using the command `pytest --cov=. tests` after installing the necessary dependencies. Our sampled projects can be executed directly with the above command, where we avoided applying complex setup scripts to avoid additional evaluation bias. The code coverage of the projects we analyzed is summarized in Figure 6, with an average coverage of 73.69% with a standard deviation of 26.76. These findings are consistent with prior empirical research on OSS by Kochhar, Thung, Lo, and Lawall



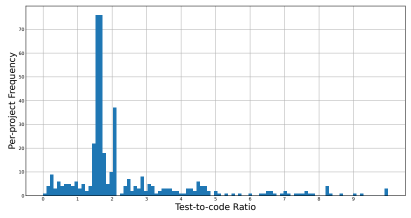
(a) Python



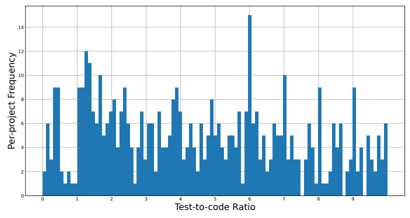
(b) Java



(c) Go

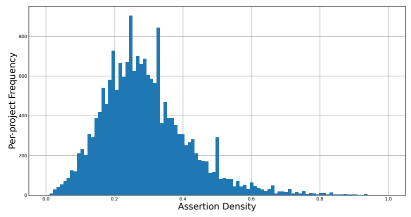


(d) C++

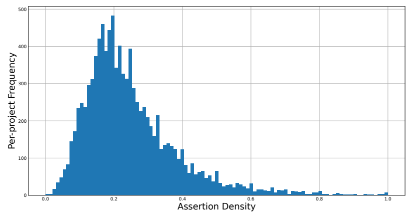


(e) JavaScript

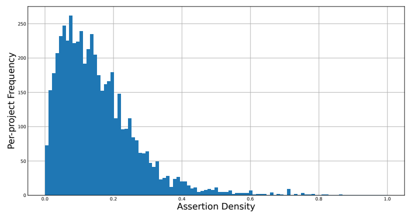
Figure 4: Distribution of per-project test-to-code ratio for all five languages in UniTSyn.



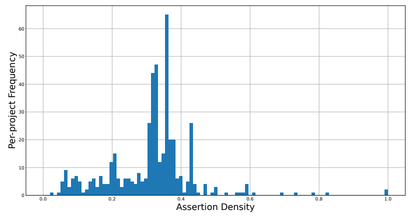
(a) Python



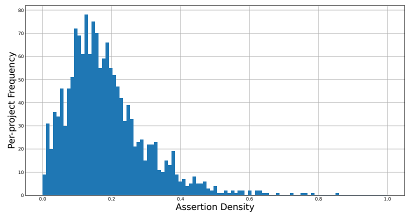
(b) Java



(c) Go



(d) C++



(e) JavaScript

Figure 5: Distribution of per-project assertion density ratio for all five languages in UniTSyn.

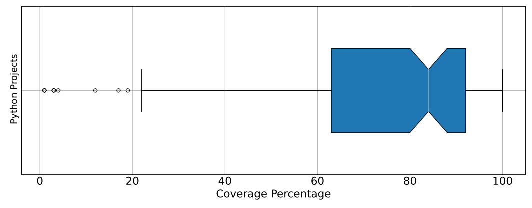


Figure 6: Coverage of projects included in UniTSyn.

and Hilton, Bell, and Marinov, reinforcing our confidence in the overall quality of the projects from which we derived our data.

3.4.4 Focal-test Alignment Accuracy. We follow previous work [43] and unit test design pattern [27] to select the last function before assertion as the call to the focal function. However, this heuristic is not perfect for finding the exact focal function. Since this requires large manual labeling, we randomly sampled 100 data for this rebuttal. We consider a pair to be aligned correctly if the function name and comments suggest so. Our data shows that 19 of them have only one function call, 59 of them have multiple function calls and our heuristic identifies the focal call correctly, 6 of them are testing for operators so there is no real focal function or matched to virtual function, and 16 of them are incorrectly identified. Therefore, our heuristic has an 84% success rate on these random samples. Although not 100% correct in alignment, there is no direct method that outperforms this heuristic since most of the test functions follow the aforementioned three-phase practice. Although the focal is not the exact match the developer intended, it is still considered partially correct due to shared semantics and is also executed and assessed in the test. We provide a detailed case study in Section 5.

For other test functions not following the common Arrange, Act, Assert paradigm described in previous work [27], we still apply the same method to find a function invocation. Even if the focal is not the exact match that the developer intended, it is still considered valid training data. This is because the matched function is called by the test function and is assessed to be working correctly. Therefore, focal-test pairs not following the Arrange, Act, Assert paradigm exactly are still beneficial to the model.

4 Experiment

We built a test generation model called **UniTester**, which was trained on UniTSyn and is capable of synthesizing unit tests for programs in different languages. To investigate the quality of our testing code corpus, we evaluated UniTester and several up-to-date code or test generation models on HumanEval-X [69], as a popular multilingual program synthesis benchmark that shares the same programming languages involved in UniTSyn.

4.1 UniTester: A Unified Test Generation Model

We constructed our model based on established code generation practices [49]. We utilized an autoregression signal [49, 55, 45] for continual training of SantaCoder [2], which is a powerful yet lightweight state-of-the-art code generation model composed of 1.1B parameters. We selected this model for its size, as this allows it to strike a balance between effectiveness and efficiency. To build a training sample, we concatenated each focal-test function pair with a newline symbol. We packed the training corpus and sampled sequences of a constant length to feed into the models, thereby avoiding padding samples of varying lengths and enhancing training efficiency. We set the constant sequence length at 2048 with a batch size of 32. Our training process employed a learning rate of $5e^{-5}$, incorporating a logarithmic warmup for the initial 500 steps and a cosine annealing strategy [37] for the rest. We also adopted a weight decay of 0.05 to refrain from overfitting and catastrophic forgetting [29]. Our UniTester was trained by an Adam optimizer [28] for $36,000$ steps on the collected data containing around one billion tokens. The entire training process using eight Nvidia V100 GPUs spanned approximately 24 hours.

4.2 Research Questions and Evaluation Setup

To study the contributions of our collected testing code corpus, UniTSyn, we investigated the following four questions:

RQ.1 How accurate are the test cases generated by LLMs? Considering that the primary objective of software testing is to identify potential flaws in code, it is essential to ensure that the generated test cases are accurate to minimize incorrect evaluations. This requires models to not only comprehend the general semantics of focal functions but also reason about their specific behavior and precise input-output mappings, which is fundamentally challenging. In this case, we followed Chen, Zhang, Nguyen, Zan, Lin, et al. to parse the assertions from the generated tests to examine their standalone correctness and derive conclusions without mutual influences.

RQ.2 How many of the generated tests are complete? We define the completeness of tests in terms of both their executability and the proportion of code in the focal functions that has actually been executed by them. To evaluate how complete the tests generated by LLMs are, we took the raw outputs of models without intricate post-processing as individual tests and reported the line/statement and branch coverage rates they achieved on the corresponding focal function. This assessment allows us to better understand the effectiveness of LLM-generated tests in covering various conditions and scenarios within the codebase without excessive manual intervention.

RQ.3 Is it necessary to train LLMs with pairwise focal and test functions? UniTSyn is designed to identify test functions and pair them with their targets in complex real-world software projects. Given that the composition of test functions heavily depends on the expected behavior of their target focal functions, we were motivated to train using test-focal function pairs. To justify our motivation, we broke the pairwise connections and treated test and focal functions as independent code snippets for LLMs training. The resulting models were then compared with our UniTester that was trained with pairwise data.

RQ.4 What are the effects of training with multilingual testing code? While previous work [15, 20] have shown that training language models on data of different distributions can be beneficial, it was unclear whether this conclusion also held for test generation. Therefore, we compared models solely trained on Python to UniTester that was trained on five different languages. This comparison aimed to determine whether test generation models should be language-specific or if they can be universal across languages.

4.2.1 *Evaluation Datasets.*
we evaluated UniTester and the state-of-the-art code LLMs on HumanEval-X [69], which is a popular and multilingual code generation benchmark dataset. HumanEval-X represents an extensive dataset encompassing 164 coding exercises/tasks, each accompanied by their respective natural language descriptions, manually composed solutions, and unit tests in Python, C++, Java, JavaScript, and Go. The conventional evaluation protocol for code generation models involve supplying the models with the signature of the intended functions, coupled with the language description to suggest the expected outcomes. Following this, the solutions generated by the models are concatenated with the handcrafted unit tests for execution. If these executions proceed without any errors, the code solutions produced by the models are deemed correct. In contrast, our evaluation prompts the models to generate tests using the solutions provided for tasks, rather than synthesizing the programs based on their problem descriptions. Subsequently, these canonical solutions are concatenated with the tests generated for execution. This feature of the HumanEval-X benchmark guarantees that the given version of the focal functions is correct, allowing us to explore LLM's ability to generate correct and useful test cases without concern about bugs in the prompt. If the executions are successful, we infer that the generated tests are correct.

We chose to experiment on an existing code generation benchmark instead of a random split of our collected test data to mitigate potential data leakage and facilitate a more straightforward execution setup. Previous work [50, 43, 56] on test generation typically validate their models on code data collected from Github or other public resources. Given that the success of the latest code LLMs is also based on large-scale open-source coding resources, it is challenging to avoid data leakage in such cases. Consequently, we turned to the benchmark datasets intended for this purpose, ensuring that all focal and test functions used for assessment had never been seen during any stages of model training.

We further ensured there exists no benchmark leakage via two methods. First, we ran a pairwise string comparison between the benchmark functions and the functions we collected from GitHub repositories to make sure there was no match. Furthermore, we manually checked repositories related to HumanEval and HumanEval-X on GitHub with more than 10 stars (as a filter requirement stated in Section 3.2). We found that all the committed datasets had text-file extensions such as `txt`, `json`, or `jsonl`, which were ignored by our dataset construction method. Therefore, we can ensure there is no benchmark leakage at our fine-tuning stage for the fairness of our evaluation.

Moreover, HumanEval-X is also well crafted such that its test cases can be executed without excessive efforts in setting up the execution environments and sorting out the intricate dependencies between packages. This allows for a more straightforward and unbiased evaluation of the models' performance in generating accurate and comprehensive tests for software projects.

4.2.2 Instructive Prompts. Following CodeT's approach [6], we prompted the models to generate tests using language-specific assert keywords. Examples of prompts for the first task in HumanEval-X are shown in Figure 7. Specially, for models trained by instruction tuning, we concatenated the focal function and our natural language hint ("`Check the correctness of...`") as the instruction then asked the models to complete the test functions. The maximal input length was set to 800 and the synthesized outputs were allowed to have another 256 tokens at most. We set the temperature for generation to 0.2 following [64, 50] and kept it consistent for all models, then sampled ten outputs for each task. Subsequently, we parsed and sampled the first ten unique assertions from the generated tests by splitting them into sub-strings using the assertion keywords as the separator [64], to compute accuracy on each task. This ensures that different models produce a similar number of assertions for verification.

4.2.3 Evaluation Metrics. Let N denote the total number of assertions parsed from the outputs of a model, we execute the assertions one at a time and count how many of them can proceed without any error as N'. Specifically, an assertion is considered successful if its execution has the returns code 0. Then, we calculate the accuracy as:

$$Acc = \frac{N'}{N}.$$

Regarding the evaluation of completeness, we treated each output of the models as an individual test and executed it independently. We avoided applying intricate post-processing on all of the models' outputs except for the Java ones, on which we found that adding closing brackets for the test class is helpful to most models for ensuring executability. To be specific, we report a "#Pass" metric to show how many tests can proceed without error during executions, and the "Line" coverages to indicate how many lines of code were executed by the tests over the total number of lines. Similarly, "Branch" coverages are computed by the number of branches traversed when executing tests over the total number of branches in the code. The average results of ten trials are reported in Table 4.

```
from typing import List
def has_close_elements(
  numbers: List[float], threshold: float) -> bool:
    ...
# Check the correctness of `has_close_elements`
def test_has_close_elements():
    assert has_close_elements(
```

Python

```
func HasCloseElements(
  numbers []float64, threshold float64) bool
{    ...    }
// Check the correctness of `HasCloseElements`
func TestHasCloseElements(t *testing.T) {
    assert := assert.New(t)
    assert.Equal(HasCloseElements(
```

Go

```
const hasCloseElements = (numbers, threshold) => {...}
// Check the correctness of `hasCloseElements`
const testHasCloseElements = () => {
    console.assert(hasCloseElements(
```

JavaScript

```
bool has_close_elements(
  vector<float> numbers, float threshold)
{    ...    }
// Check the correctness of `has_close_elements`
#undef NDEBUG
#include <assert.h>
int main(){
    assert(has_close_elements(
```

C++

```
class Solution {
    public boolean hasCloseElements(
      List<Double> numbers, double threshold)
    {    ...    }
}
public class Main {
    // Check the correctness of `hasCloseElements`
    public static void main(String[] args) {
        Solution s = new Solution();
        assert s.hasCloseElements(
```

Java

Figure 7: Prompts for test generation in different languages

4.2.4 Compared Models. We conducted extensive comparisons between our proposed model and both the state-of-the-art code generation models highlighted in Xiong, Guo, and Chen's paper as well as the latest test generation model [50]. In our evaluation, we considered code generation models with a similar number of parameters to ours, encompassing both encoder-decoder (CodeT5+ [61]) and decoder-only structures (CodeGen2 [44], WizardCoder [39], InCoder [17] and SantaCoder [2]). For test generation, we selected CAT-LM [50] as our competitor, which was trained on unit tests from Python and Java projects that were paired with their target functions at the file level. CAT-LM was chosen because it is trained

Table 3: Accuracy of tests generated by LLMs. The best results are highlighted in bold.
#Params: the size of models
†: the models are intended for test generation.

Model	#Params	Py	C++	Java	JS	Go	Avg
CodeT5p	770M	30.6	33.7	26.9	37.1	32.9	32.2
CodeGen2	1.0B	34.0	40.7	24.1	30.5	36.1	33.1
WizardCoder	1.0B	36.8	43.9	28.7	31.3	47.7	37.7
InCoder	1.3B	34.2	33.5	22.6	24.4	31.5	29.2
SantaCoder	1.1B	36.2	34.7	36.5	30.6	31.5	33.9
CAT-LM†	2.7B	37.5	31.6	34.4	29.2	36.9	33.9
UniTester† (Ours)	1.1B	**52.5**	**55.1**	**48.8**	**41.7**	**59.7**	**51.5**

on a much larger number of tokens (60B *v.s.* 1B) than our model and has double our model size. This allowed us to demonstrate our model's superior performance, despite any potential advantages stemming from the amount of training data and the model size.

4.3 Evaluation Results

4.3.1 RQ.1 How accurate are the test cases generated by LLMs? The accuracy of the test cases generated by different models is presented in Table 3. Our model achieved up to a 40% relative margin on Python compared to the top competitor (CAT-LM), demonstrating remarkable advantages over both code and test synthesis models. In terms of the code generation models, UniTester beat the strongest competitor (WizardCoder) by 36.6% on average. These results validate the quality of the data in UniTSyn and indicate its benefits on enhancing LLMs' capability of in-depth code understanding and reasoning in order to ensure the high accuracy of the generated test cases. Our improvements over the baseline SantaCoder [2] on the two languages that it was not trained on (C++ and Go) are on par with the rest. The advantages are at times even more significant on languages with insufficient training data, e.g., JavaScript. This implies that it might not be necessary to have our base models trained on every language of interest to generate high-quality tests. In addition, we observed that our improvements over the top competitors across different languages are somewhat dependent on the sufficiency of the tests collected, with the least improvement being in JavaScript and the most in Python. This observation highlights the contribution of UniTSyn, which is capable of collecting testing data for different languages at scale.

4.3.2 RQ.2 How many of the generated tests are complete? We presented the average number of passing tests generated by different models and the average coverage rates they achieved on the focal functions in Table 4. The passing rates of each model across all languages range from 2% to 32%. Low passing rates indicate the necessity of both accurate and executable test functions generated by LLMs. Our UniTester demonstrated considerable advantages in this regard, with its average passing number being nearly twice that of the top competitor, CodeGen2 [44]. This result shows the inconsistency between the distributions of focal functions and tests, emphasizing the necessity of training models with a high-quality testing code corpus. We also observed that composing executable tests for C++ was the most challenging among all the languages. This is not surprising, as C++ is considered relatively weaker in readability and harder to code in even for human developers. Moreover, UniTester yielded superior coverage rates, with absolute improvements of up to 6.32% and an average of 2.8% regarding line coverage. Such improvements underscore the effectiveness of UniTester in covering various conditions in the focal functions.

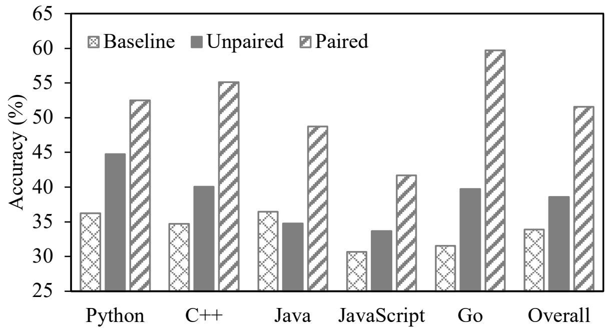


Figure 8: Impact of pairing test and focal functions. Baseline: the SantaCoder model, not trained with our data. Unpaired: trained with decoupled test and focal functions. Paired: UniTester trained with focal-test pairs.

4.3.3 RQ.3 Is it necessary to train LLMs with pairwise focal and test functions? In Figure 8, we constructed an "Unpaired" variant of UniTester ("Paired") by decoupling test functions from their targets for model training and compared it with our models trained with pairwise focal-test data. The unpaired variant of UniTester demonstrated its capability to generate more accurate tests than the baseline. While SantaCoder's paper did not explicitly state the exclusion of testing code from their training corpus, it is likely that their model was also trained with a certain number of test functions. However, it is shown to be less competent for test generation than the "Unpaired" model. One possible reason for this is the more balanced focal-test functions in UniTSyn, despite being unpaired. Furthermore, the remarkable performance advantages of UniTester over its unpaired counterpart indicate the importance of associating test functions with their targets. Including focal functions in the context when generating tests explicitly provides the models with insights into the expected usages and behavior. Without these insights, the model can only learn to make reasonable predictions for test functions by memorizing all possible focal functions, rather than through reasoning. This observation highlights the value of UniTSyn on not only collecting testing data but also matching it with the focal functions in a language-agnostic manner.

4.3.4 RQ.4 What are the effects of training with multilingual testing code? The construction of all-purpose code generation models has been garnering increased attention. Previous studies have also suggested the potential benefits of training on data with shared semantics but different distributions [15]. Our proposed UniTSyn is extendable to collect tests for any programming language with an available language server, enabling us to build a universal test generation model. To investigate the best practice of utilizing a multilingual testing code corpus for training LLMs, we compared UniTester with its monolingual variant, which was trained solely on

Table 4: Completeness of LLM-generated tests.
#Params: size of the model. #Pass: percentage of tests for the 164 tasks that can be executed without errors.
Line, Stmt, Branch: average line, statement, and branch coverage, respectively.
†: the model is intended for test generation.
Limited by the coverage evaluation tools we adopted, branch coverage is not available on C++ and Go.

Model	#Params	Python			C++		Java			Javascript			Go	
		#Pass	Line	Branch	#Pass	Line	#Pass	Line	Branch	#Pass	Line	Branch	#Pass	Stmt
CodeT5p	770M	10.0	5.72	5.41	0.7	0.43	40.3	4.22	2.01	4.9	2.07	1.01	1.7	0.73
CodeGen2	1B	4.1	2.41	2.34	11.6	7.07	52.3	5.12	3.29	48.5	27.65	23.87	19.2	10.99
WizardCoder	1B	16.1	9.39	8.95	3.7	2.24	47.7	5.62	4.09	9.2	5.50	5.32	0.7	0.42
InCoder	1.3B	3.0	1.76	1.60	0.0	0.00	15.0	1.54	0.91	0.5	0.29	0.26	1.3	0.78
SantaCoder	1.1B	4.5	2.62	2.59	4.9	2.99	50.1	4.74	1.83	5.9	3.53	3.23	0.7	0.43
CAT-LM†	2.7B	35.9	19.51	18.03	0.0	0.00	0.9	0.07	0.00	9.2	4.53	3.49	0.0	0.00
UniTester† (Ours)	1.1B	41.2	20.71	18.27	28.1	13.39	103.1	10.78	4.57	53.3	27.59	23.34	36.0	12.39

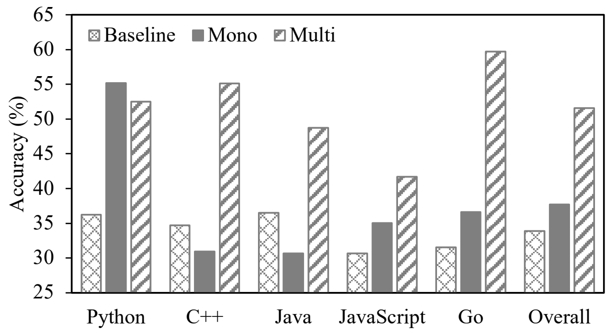


Figure 9: Effects of training with multilingual testing code. Baseline: the SantaCoder model, not trained with our data. Mono: monolingual model trained with solely Python data. Multi: multilingual models trained jointly with five languages.

the Python subset of UniTSyn. We denoted UniTester and its monolingual variant by "Multi" and "Mono" respectively in Figure 9. The figure reveals that the monolingual model demonstrated superior capability in generating high-quality tests for Python. This capability transferred well to other scripting languages like JavaScript and Go, but less so to C++ and Java. These results suggest modest transferability between syntactically similar languages but also indicates the potential for negative impacts elsewhere. Regardless of whether the aim is to build test generation models for specific languages or for general purposes, a flexible and universal framework for collecting tests from different languages is indispensable. This demonstrates the contribution of our proposed UniTSyn in the field of software testing.

5 Limitations

Hooks for Different Unit Testing Framework. Despite our effort to design UniTSyn to involve as little human effort as possible, some manual settings are inevitable. For example, C++ and JavaScript do not have a commonly used testing framework, meaning that the developer has to implement the hook to identify test functions for each framework. Since we only implemented the test function identification hook for the GoogleTest suite, this accounts for our low repository number in C++. We believe one can extract more focal-test pairs from the repositories by applying more precise hooks to our framework. Our goal is to build an extensible multilingual dataset instead of diving deep into different C++ and JavaScript testing suites.

```python
def multiple(a: int, b: int):
    return a * b
def is_even(number: int):
    return number % 2 == 0
def test_multiplication():
    a: int = random.randint(0, 100)
    mul_res: int = multiply(a, 2)
    prop_res: bool = is_even(mul_res)
    assertTrue(prop_res)
```

Listing 2: A Python property-based testing as partially correct paired by UniTSyn's heuristic

Focal Call Selection Heuristics. We randomly sampled 100 examples to evaluate the accuracy of this focal-test pairing heuristic in Section 3.4.4. Our heuristic achieved 84% accuracy. Even though some of the focal functions are not the exact match, they share close semantics or functionality with the real focal function as a wrapper or a related member function of the same class. In this section, we provide some detailed case studies on the failed cases and discuss possible directions for future work.

First, even if the focal is not the exact match the developer intended, it is still considered partially correct. This is because the match function is called by the test function and is assessed to be working correctly. Taking the `MathOperations` class in Listing 2 as an example, though the test is named `testMultiplication`, its implementation also checks if `MathOperations.isEven` works correctly. Therefore, it can also be considered as a test function for `MathOperations.isEven`. This example is also a practice of property-based testing [11] but without generating multiple cases from the input space, which is not in the scope of UniTSyn.

Second, some testing functions may receive the object under test from its parameter instead of constructing it in its function body. For example in Listing 3, `pytest` allows users to parametrize

```python
def has_attribute(self, attribute: str)->bool:
    return any([(key_node.value == attribute) for key_node, _
          in self.yaml_node.value])
def test_dashes_to_unders_in_keys(class_node: yatiml.Node)->
    None:
    assert class_node.has_attribute('dashed-attr')
    class_node.dashes_to_unders_in_keys()
    assert class_node.has_attribute('dashed_attr')
    assert class_node.has_attribute('list1')
```

Listing 3: A Python parametrized test

```python
def _makeOne(self, system, val=None):
    from pyramid.events import BeforeRender
    return BeforeRender(system, val)
def test_setdefault_fail(self):
    event = self._makeOne({})
    result = event.setdefault('a', 1)
    self.assertEqual(result, 1)
```

Listing 4: A Python test for imported function

tests [46]. In this case, no program analysis based heuristic can compute the correct focal function given solely the test function. The correct focal function for `test_dashes_to_unders_in_key` would be the class constructor of `yatiml.Node`.

The third frequent mismatched case is testing for imported library functions. When analyzing our dataset, we found that some test functions make assertions for the output of library functions. For example in Listing 4, `test_setdefault_fail` is testing the `setdefault` function. Our heuristic only matches focal functions defined in the project's codebase, whereas `setdefault` is a class method of `BeforeRender`, which is imported from `pyramid`.

In our experiments, we have found identifying the last function call before an assertion is the most direct and reasonable heuristic. Nevertheless, there is room for further improvement in this. One potential method involves counting function calls to determine the most frequently called function as the focal point. Beyond the AST level of program abstraction, the Control Flow Graph (CFG) of the program could also be utilized for this purpose. For instance, combining a random walk on the CFG with weighted paths [12] might be a viable approach. However, these methods are not without flaws. The complex heuristics pose significant challenges in generalization across different programming languages, which is essential for building a multilingual and diverse dataset like UniTSyn.

6 Conclusion

In this paper, we present UniTSyn, a novel, diverse, and large-scale dataset containing function-level focal-test pairs designed to stimulate AI in understanding and writing programs, particularly for test cases. This dataset not only excels in size and diversity, but is also effortlessly extendable to other programming languages for specific tasks. We further built an autoregressive model based on UniTSyn to verify the quality of the collected testing code corpus. This is shown by its superiority in terms of both the accuracy and completeness of the generated tests. We demonstrate that UniTSyn can be useful for the development of AI for software testing and program understanding in general. Our code and data are available [60].

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1801751 and 1956364, and by UC Noyce Initiative. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or UC.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, (July 2020). https://doi.org/10.18653/v1/2020.acl-main.449.

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

[3] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40, 11, 1100–1125.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2015). Association for Computing Machinery, Bergamo, Italy, 179–190. isbn: 9781450336758. https://doi.org/10.1145/2786805.2786843.

[5] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert use in github projects. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2015.88.

[6] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, et al. 2023. Codet: code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating large language models trained on code. (2021). arXiv: 2107.03374 `[cs.LG]`.

[8] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 711–725. https://doi.org/10.1109/SP.2018.00046.

[9] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (CCS '19). Association for Computing Machinery, London, United Kingdom, 499–513. isbn: 9781450367479.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of jvm implementations. *SIGPLAN Not.*, 51, 6, (June 2016), 85–99. https://doi.org/10.1145/2980983.2908095.

[11] Koen Claessen and John Hughes. 2000. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (ICFP '00). Association for Computing Machinery, New York, NY, USA, 268–279. isbn: 1581132026. https://doi.org/10.1145/351240.351266.

[12] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, 423–433. https://doi.org/10.1145/3236024.3236059.

[13] 1972. *Chapter i: notes on structured programming. Structured Programming.* Academic Press Ltd., GBR, 1–82. isbn: 0122005503.

[14] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. Toga: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22). Association for Computing Machinery, Pittsburgh, Pennsylvania, 2130–2141. isbn: 9781450392211. https://doi.org/10.1145/3510003.3510141.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, et al. 2020. CodeBERT: a pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 10–10.

[17] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, et al. 2022. Incoder: a generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

[18] [n. d.] Gopter: the golang property tester. https://github.com/leanovate/gopter.

[19] Nadeeshaan Gunasinghe and Nipuna Marcus. 2021. *Language Server Protocol and Implementation*. Springer. Chap. 8.

[20] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, et al. 2022. Unixcoder: unified cross-modal pre-training for code representation. In *ACL*.

[21] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: a study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 38–47.

[22] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 53–63.

[23] Jiabo Huang, Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, et al. 2023. Code representation pre-training with complements from program executions. (2023). arXiv: 2309.09980 [cs.SE].

[24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: evaluating the state of semantic code search. (2020). arXiv: 1909.09436 [cs.LG].

[25] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Berlin, Germany, (Aug. 2016). https://doi.org/10.18653/v1/P16-1195.

[26] [n. d.] Junit-quickcheck: property-based testing, junit-style. https://pholser.github.io/junit-quickcheck/site/1.0/.

[27] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

[28] Diederik P Kingma and Jimmy Ba. 2014. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[29] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114, 13, 3521–3526.

[30] Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, et al. 2023. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

[31] Pavneet Singh Kochhar, Ferdian Thung, David Lo, and Julia Lawall. 2014. An empirical study on the adequacy of testing in open source projects. In *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. IEEE, 215–222.

[32] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. 2006. Assessing the relationship between software assertions and faults: an empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, 204–212. https://doi.org/10.1109/ISSRE.2006.14.

[33] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[34] Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, et al. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*.

[35] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, et al. 2022. Competition-level code generation with alphacode. *Science*, 378, 6624, 1092–1097. https://www.science.org/doi/abs/10.1126/science.abq1158 eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

[36] Ziheng Liu, Shufei Zhu, Boqin Qin, Hao Chen, and Song Linhai. 2021. Automatically detecting and fixing concurrency bugs in go software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. (Apr. 19–23, 2021).

[37] Ilya Loshchilov and Frank Hutter. 2016. SGDR: stochastic gradient descent with restarts. *CoRR*. http://arxiv.org/abs/1608.03983.

[38] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, et al. 2021. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[39] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, et al. 2024. Wizardcoder: empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

[40] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: a new approach to property-based testing. *Journal of Open Source Software*, 4, 43, 1891.

[41] Microsoft. 2024. Language server protocol. (Jan. 11, 2024). https://microsoft.github.io/language-server-protocol/.

[42] Nachiappan Nagappan and Thomas Ball. 2010. Evidence-based failure prediction. In *Making Software*. Greg Wilson Andy Oram, (Ed.) O'REILLY. Chap. 23.

[43] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2111–2123.

[44] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: lessons for training llms on programming and natural languages. In *The Eleventh International Conference on Learning Representations*.

[45] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, et al. 2023. Codegen: an open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

[46] [n. d.] Parametrizing tests. https://docs.pytest.org/en/8.0.x/example/parametrize.html.

[47] [n. d.] Property based testing framework for javascript/typescript. https://github.com/dubzzz/fast-check.

[48] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, et al. 2021. Codenet: a large-scale ai for code dataset for learning a diversity of coding tasks. In *Neural Information Processing Systems (NeuralIPS)*.

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1, 8, 9.

[50] 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 *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[51] [n. d.] Rapidcheck: quickcheck clone for c++ with the goal of being simple to use with as little boilerplate as possible. https://github.com/emil-e/rapidcheck.

[52] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. 2021. The specification language server protocol: a proposal for standardised lsp extensions. In *Proceedings of the 6th Workshop on Formal Integrated Development Environment*, 3–18.

[53] Yuyang Rong, Peng Chen, and Hao Chen. 2020. Integrity: finding integer errors by targeted fuzzing. In *International Conference on Security and Privacy in Communication Systems*. Springer, 360–380.

[54] Yuyang Rong, Chibin Zhang, Jianzhong Liu, and Hao Chen. 2024. Valkyrie: improving fuzzing performance through deterministic techniques. *Journal of Systems and Software*, 209, 111886.

[55] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, et al. 2023. Code llama: open foundation models for code. (2023). arXiv: 2308.12950 [cs.CL].

[56] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

[57] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, 157–157. https://doi.org/10.1109/SecDev.2016.043.

[58] [n. d.] Tree-sitter introduction. https://tree-sitter.github.io/tree-sitter/.

[59] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context. (2021). arXiv: 2009.05617 [cs.SE].

[60] 2024. UniTSyn. https://doi.org/10.5281/zenodo.12639546.

[61] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, et al. 2023. Codet5+: open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 1069–1088.

[62] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH 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*, 8696–8708.

[63] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, (June 2020). https://doi.org/10.1145%2F3377811.3380429.

[64] Weimin Xiong, Yiwen Guo, and Hao Chen. 2023. The program testing ability of large language models for code. *arXiv preprint arXiv:2310.05727*.

[65] Qian Yang, J. Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test* (AST '06). Association for Computing Machinery, Shanghai, China, 99–103. ISBN: 1595934081. https://doi.org/10.1145/1138929.1138949.

[66] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, et al. 2024. Codereval: a benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.

[67] Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Understanding programs by exploiting (fuzzing) test cases. In *Findings of the Association for Computational Linguistics (ACL)*. Toronto, Canada. https://doi.org/10.18653/v1/2023.findings-acl.678.

[68] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE.2017.8115619.

[69] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, et al. 2023. Codegeex: a pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5673–5684.

[70] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the reproducibility of software defect datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2324–2335.

[71] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29, 4, (Dec. 1997), 366–427. https://doi.org/10.1145/267580.267590.

Received 2024-04-12; accepted 2024-07-03