Domain Adaptation for Code Model-Based Unit Test Case Generation

Jiho Shin
jihoshin@yorku.ca
York University
Toronto, Ontario, Canada

Sepehr Hashtroudi
sepehr.pourabolfathh@ucalgary.ca
University of Calgary
Calgary, Alberta, Canada

Hadi Hemmati
hemmati@yorku.ca
York University
Toronto, Ontario, Canada

Song Wang
wangsong@yorku.ca
York University
Toronto, Ontario, Canada

ABSTRACT

Recently, deep learning-based test case generation approaches have been proposed to automate the generation of unit test cases. In this study, we leverage Transformer-based code models to generate unit tests with the help of Domain Adaptation (DA) at a project level. Specifically, we use *CodeT5*, a relatively small language model trained on source code data, and fine-tune it on the test generation task. Then, we apply domain adaptation to each target project data to learn project-specific knowledge (project-level DA). We use the *Methods2test* dataset to fine-tune *CodeT5* for the test generation task and the *Defects4j* dataset for project-level domain adaptation and evaluation. We compare our approach with (a) *CodeT5* fine-tuned on the test generation without DA, (b) the *A3Test* tool, and (c) *GPT-4* on five projects from the *Defects4j* dataset. The results show that tests generated using DA can increase the line coverage by 18.62%, 19.88%, and 18.02% and mutation score by 16.45%, 16.01%, and 12.99% compared to the above (a), (b), and (c) baselines, respectively. The overall results show consistent improvements in metrics such as parse rate, compile rate, BLEU, and CodeBLEU. In addition, we show that our approach can be seen as a complementary solution alongside existing search-based test generation tools such as *EvoSuite*, to increase the overall coverage and mutation scores with an average of 34.42% and 6.8%, for line coverage and mutation score, respectively.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**.

KEYWORDS

Test generation, Transformers, LLM, GPT, Code Model, Domain Adaption

ACM Reference Format:
Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain Adaptation for Code Model-Based Unit Test Case Generation. In *ISSTA '24: ACM SIGSOFT International Symposium on Software Testing and Analysis, September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Code models that are pre-trained on a large corpus of source code have been introduced to automate numerous software development tasks such as comment generation, code translation, and code generation [3, 17, 34, 35]. Among these downstream tasks, unit test generation, which can be seen as a neural machine translation task, has started gaining its spotlight recently [4, 39].

There are several reasons for the challenges in unit test case generation: (a) The robustness of the code generation model is more challenging to achieve, as slight miss generation would lead to an error. Unit test case generation, in particular, might be more challenging than regular code generation as test cases tend to have more minor differences between code. For example, a line of assertion statements or a couple of statements to instantiate objects might drive the program into an interesting and testable state. (b) Properly evaluating the generated test cases requires executing the generated tests to calculate test adequacy metrics, which is time-consuming and typically requires non-trivial manual labor, e.g., resolving dependencies. (c) Domain shift problem [46] occurs when the pre-trained models cannot transfer their code knowledge to a new target project due to different code distributions in various domains of projects.

Despite these shortcomings, test case generation based on deep neural code models has advantages. The generated tests from neural models are similar since the models are trained on human-written code. Therefore, they are more readable and maintainable than the alternative automatically generated test cases. As previous literature suggests [39], developers prefer neural model-generated tests over other automatically created test cases since they are more readable and understandable. They also target different faults (the same as those targeted by the developer-written tests) compared to tests generated by, e.g., search-based approaches, which usually focus on maximizing code coverage.

To address the shortcomings of pre-trained code models for test case generation, i.e., low performance, insufficient evaluation, and

ISSTA 2024, 16-20 September, 2024, Vienna, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

domain shift, we propose a simple yet novel technique by adopting two different levels of fine-tuning/domain adaptation: task and project. In our approach, first, we fine-tune the *CodeT5* pre-trained model with a task-specific dataset to customize the model for generating unit test cases, given a method under test. Then, we apply domain adaptation with the project-specific dataset to learn the proper code knowledge and create higher-quality test cases for mitigating the impact of the domain shift problem. We also conduct a more thorough investigation by evaluating test adequacy and textual similarity metrics to address the insufficient evaluation problem. Regardless of the simplicity of the idea, we note that this approach is 1) novel and 2) effective as it enables the relatively smaller model (*CodeT5* with 220M parameters) to outperform much bigger models (*GPT-4* with 1.76T). Our framework uses automated post-processing of simple heuristics to mitigate compilability/executability issues. We use the *Methods2test* dataset [38] for fine-tuning the test case generation task. We apply domain adaptation to the models by leveraging human-written unit test cases for each project. For evaluation and domain adaptation, we use the *Defects4j* dataset [16]. We compare the effectiveness of our approach with and without domain adaptation. We also investigate two other baselines, namely *GPT-4* (the largest and the state-of-the-art LLM) and *A3Test* (state-of-the-art neural test case generation method which exploits task-knowledge domain adaptation). Our model with project-level domain adaptation outperforms all the baselines on all the studied metrics, except for the parse rate of *GPT-4*. Furthermore, our approach can be used alongside search-based test generation to increase their line coverage and mutation score.

We show that using domain adaptation, we can improve the line coverage with an average of 18.62%, 19.88%, and 18.02% and mutation score by 16.45%, 16.01%, and 12.99% compared to *CodeT5* without DA, *A3Test*, and *GPT-4* baselines, respectively. We also show that our approach can increase the overall coverage and mutation scores of *EvoSuite* when used alongside each other, with an average of 34.42% and 6.8% for line coverage and mutation score, respectively.

In summary, our main contributions are as follows:

(1) We propose a line-level neural test case generation framework leveraging domain adaptation, which creates high-quality unit test cases (compilable, similar to human-written, and test-adequate).
(2) We conducted an empirical study on *Defects4j* benchmark dataset [16], which shows our approach improves the performance of the most related work *AthenaTest*, *A3Test*, and *GPT-4*) from the literature.
(3) We also show that our approach can cover lines that neither developer-written tests nor a baseline search-based testing tool can cover. We also showed that we can kill new mutants compared to the search-based tools.
(4) Unlike most related work, we execute the generated test cases and evaluate them with proper test adequacy metrics (i.e., code coverage and mutation score), which require much more effort to calculate compared to BLEU/CodeBLEU. We also report the BLEU and CodeBLEU scores, which are much used in the literature for automated evaluation metrics.

The code for our proposed approach and the experiment's scripts and raw data are publicly available for replication[1].

We organized the rest of this paper as follows. Section 2 introduces the background of neural models for code and unit test generation. Section 3 presents the approach of our test case generation framework. Section 4 shows the experimental setup. Section 5 presents the evaluation results. Section 6 discusses the possible threats in our study. Section 7 concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 Search-based Software Testing

In search-based software testing (SBST), the problem of test case generation is translated into an optimization problem over a test adequacy criterion such as code coverage [21]. For instance, *EvoSuite* [11] is an SBST tool that generates test cases to optimize statement or branch coverage of the generated tests. It uses a genetic algorithm to evolve a test suite toward a higher quality set (more coverage with minimum tests). While SBST tools have shown great effectiveness, studies report their limitation in understandability or readability [6, 14, 30], quality [13, 24], and their performance in detecting actual bugs from the generated unit test cases. [5, 27]

2.2 Domain Adaption

Domain adaptation is a technique for modifying a model trained on one domain to perform well on a different but related domain. The goal is to leverage the knowledge gained from the source domain to improve the performance of the target domain, mainly when the target domain has limited labeled data. Domain adaptation is a type of transfer learning which aims to transfer knowledge from one task to another.

Nam et al. [22] proposed a novel transfer defect learning approach, *TCA+*, which applies a transfer learning technique to reduce the data distribution difference between source and target projects for cross-project defect prediction. *TCA+* also selects a suitable normalization option based on the similarity of data set characteristics between the source and target projects and significantly improves prediction performance. Patel et al. [26] did a survey about domain adaptation methods for visual recognition. The paper discusses the challenges, assumptions, and formulations of domain adaptation and categorizes the existing methods into feature augmentation, feature transformation, parameter adaptation, dictionary learning, and others. It also highlights the advantages and limitations of each category and identifies some promising directions for future research in this field. Farahani et al. [9] have briefly reviewed domain adaptation. It introduces the main categories, challenges, and domain adaptation approaches, focusing on unsupervised domain adaptation. Zirak et al. [46] propose a domain adaptation framework for automated program repair (APR) models that can improve their effectiveness on new and different projects. The framework uses three methods: full fine-tuning, tuning with lightweight adapter layers, and curriculum learning. It also employs a data synthesis method to create artificial bugs for zero-shot learning.

[1] https://github.com/shinjh0849/unit_tc_generation

2.3 Neural Models for Unit Test Generation

Deep neural models of code for unit test case generation are limited and relatively new. They can be grouped into two categories, i.e., test oracle generation and unit test case generation.

2.3.1 Test Oracle Generation. Test oracle generation aims to generate oracles, e.g., meaningful assertion statements when the focal context (method under test together with its class information, i.e., class method signature and class fields) and the corresponding test prefix are given [33]. Test prefixes are statements in a unit test case with the oracles (assertion statements, try-catch clause, etc.) removed. Test prefixes drive the program into a desired testable state. In general, the problem of oracle generation is a subset of the whole test case generation.

ATLAS (AuTomatic Learning of Assert Statements) [42] is the first to utilize deep neural models for assertion generation. They could generate assertions with the BLEU-4 score of 61.85%. Yu et al. [44] introduced an approach to integrate information retrieval techniques, using Jaccard coefficient [37], Overlap [43], and Dice coefficient [7] with the deep neural approach *ATLAS*. With their approach, they could boost the BLEU score up to 78.86%. *TOGA* (a neural method for Test Oracle GenerAtion) [8] was proposed to use a unified transformer-based neural model to generate both try-catch clause and assertion statements for unit test case oracles. For generating the try-catch clause, they had 86% of exact match accuracy and 69% for assertion statements. Tufano et al. [40] proposed to apply the *BART* pre-training model trained with natural language and source code corpus and then fine-tune on *ATLAS* dataset. They achieved an exact match accuracy of 62.47% with a beam size of one.

The main difference between our work and test oracle generation is that test oracle generation models only focus on the oracle part of the test case. Generating test prefixes is a non-trivial task, which calls into the need to generate whole unit test cases.

2.3.2 Unit Test Case Generation. There have not been many studies related to automating the generation of whole test cases. Liu et al. [20] exploited deep learning models to generate relevant text inputs to test user interfaces for mobile applications. Saes [31] generated a test suite for Java projects by identifying the connections between focal methods and their corresponding tests. They have gathered more than 780K pairs of focal and test methods utilizing the JUnit testing framework from GitHub. They could generate test cases with a parsability of 86.69%. However, they did not evaluate how correct or effective the generated test cases were in identifying bugs or covering code. Tufano et al. [39] proposed *AthenaTest* which exploits the *BART* pre-trained model on both natural language and source code corpora then fine-tune on *Methods2Test* [38] dataset, to generate whole unit test cases when a focal method and its context is given. They have found that their method could correctly test 43% focal methods, with 16% of the candidates being correct. Alagarsamy et al. [4] proposed *A3Test*, which is a test case generation approach that is augmented by a test oracle generation task and includes a mechanism to verify naming consistency and test signatures. It performs domain adaptation at a task level, i.e., test oracle generation task to whole test case generation task, achieving more correct test cases and method coverage than *AthenaTest.* Lemieux et al. [19] proposed *CodaMOSA*, an SBST approach that leverages LLM for escaping the coverage plateau for Python code bases. Schafer et al. [32] proposed *TestPilot*, a test cases generation approach that leverages LLM, usage examples mined from package documentation, and error logs for npm packages (JavaScript). Yuan et al. [45] proposed *ChatTester*, which is a LLM-based test case generation model that exploits *ChatGPT* and iterative generate-and-validate prompt engineering strategy with execution feedback. Nie et al. [23] proposed *TeCo*, a deep encoder-decoder test completion model that learns different levels of code semantics and re-ranking by execution. A test completion model generates the following statement of a unit test case when the previous line and method under test are given. Our study continues in this direction and proposes domain adaptation at a project level to improve *AthenaTest* and *A3Test* as our most related work. Unlike these papers, we evaluate based on classic software testing criteria (i.e., code coverage and mutation testing). Most existing approaches only report BLUE scores or similar NLP-based metrics that do not correlate with the effectiveness (adequacy) of the generated test cases. Although there is literature regarding test case generation, it has shown that we still have challenges in generating correct and effective test cases that reveal bugs for practical usage.

3 TEST CASE GENERATION WITH PROJECT LEVEL DOMAIN ADAPTATION

Figure 1 illustrates our proposed test case generation framework. Our approach contains two major steps: (a) fine-tuning the *CodeT5* model on a task-level dataset and then (b) applying domain adaptation (DA) on a project-level dataset. The following sections explain the two steps in detail. The framework aims to generate high-quality test cases with adequate test efficacy learned from developer-written test cases.

3.1 Fine-tuning on Test Case Generation Task

Our framework assumes the project under test has an initial test suite. We aim to improve the project by generating new tests using code models. Although we use developer-written test suites as our initial set, they can also be automatically generated (e.g., using ChatGPT). We explain each option's limitation in the threats to the validity section.

The first step is to create a **coverage database** from the existing test suite. We use the line-level coverage in our framework and evaluation for simplicity. However, this can be extended to other code metrics or mutation scores. The coverage database keeps the information on which unit test covers which lines of source code. In the next step, our **line2test mapping** approach converts the coverage data to map each line in the source code to its covering tests. **Line2test mapping** extracts the classpath of all test cases.

Next, we fine-tune the *CodeT5* model on the “test case generation” downstream task. We fine-tune the *CodeT5* model since they are not specifically trained on test generation tasks. We use “conditional code generation” proposed by the original *CodeT5* paper to optimize the model for fine-tuning. Conditional code generation generates code similar to conventional natural language processing. They adapt the conventional Sequence-to-Sequence framework for learning the task-specific data [36].

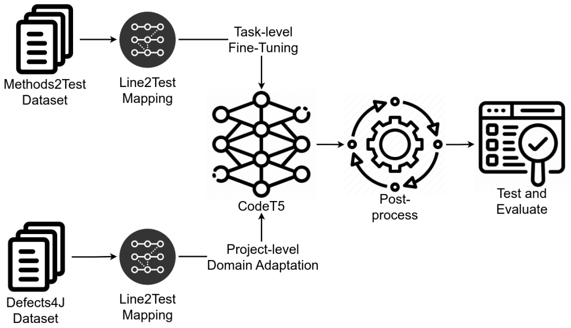


Figure 1: Overview of our approach.

The dataset to fine-tune the code model for the test generation downstream task is the *Methods2Test* dataset [38] (henceforth **test generation data**). It consists of tuples of a focal method, focal context (e.g. class name, class fields, public method signatures), and the associated unit test method. Our framework maps between the input source method, the context of the input source method, and a test method that covers the source method.

The final step in the framework is **post-processing** the test cases. Since the model accepts data input and output in one line, we replaced the new line ("\n") characters with the "[EOL]" token. The first step in post-processing is to replace "[EOL]" with "\n". Then, we make a list of generated test cases. We add a unique number at the end of the duplicate test names to prevent compile errors due to duplicate definitions of the same test case. The next step is to select compilable tests because not all the model-generated tests are compilable. We include another step to automate the inclusion/exclusion process. Since compiling all the tests is time-consuming, we first use a Java parser to select the tests without syntax errors. We use tree-sitter [2] for the implementation, a parser that supports multiple programming languages. Since the parser does not compile the code, we can identify all parsing issues in a class in less than a second. Some generated tests are truncated due to the model's token limitation, thus will result in a parse error. To make them parsable, we apply a simple bug-fix pattern to them. These tests generally follow the same pattern, i.e., they do not match brackets or have a missing ";" at the end of the last line. Using an automated script, we fix these issues by deleting the last line (usually, the test ends in the middle of a line) and adding a closing bracket. Then, we parse the test again. If it is still not parsable, we add another closing bracket and re-parse. After selecting the final parsable tests, we add each test to its corresponding test class to compile them in the project environment. We use the classpath for each test from the **line2test mapping** step. The test class has the required dependencies and other test helpers that the test case may need to be compilable. We add each test case to the corresponding file and compile it individually to ensure it's also compilable. Some tests may not pass this step for several reasons, such as calling undefined or unreachable objects or functions. To fix these compile issues, we add the test cases to their corresponding test class file consisting of test helpers and other dependencies (saved in the line2test step).

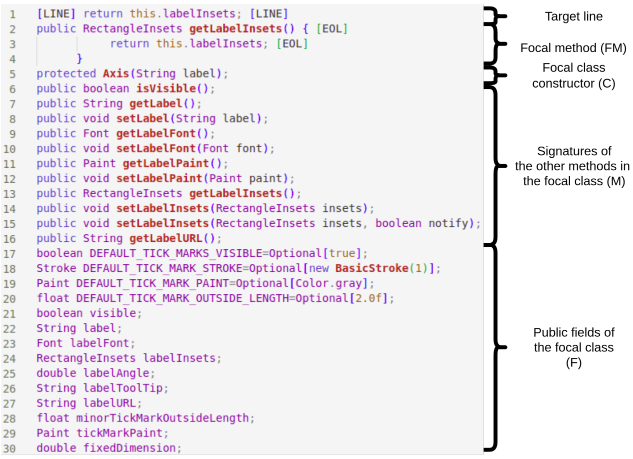
Finally, we remove all the developer-written tests and add each model-generated unit test to its corresponding test class. Even with one test with a compilation error, the build would not succeed. Thus, we add one test case at a time and compile the project using the *Defects4j* framework. If the test is compilable, we add it to the list of parsable and compilable tests; if it does not compile, we exclude it from our test set. Using this fully automated post-processing step, we now have a set of compilable model-generated test cases.

3.2 Domain Adaptation on Project Specific Knowledge

The main downside of a test case generation model is its inability to adapt to potential domain shifts when the model is inferring a new project. This phenomenon (domain shift) is not limited to the test case generation task and applies to all machine learning tasks. The structural difference in each project may cause a drastic change in the context generated by the framework making it harder for the model to generate compilable tests. We leverage the existing developer-written tests for each project to mitigate this threat. Usually, a well-maintained project already has a test suite covering most of the code. We use the existing test suite to generate a project-specific dataset for domain adaptation. Alternatively, one can start with a set of automatically generated test cases and further improve them with our approach.

As demonstrated in Figure 1, the first step for applying domain adaptation is to generate a dataset using developer-written tests. Since a target line can be as simple as a return statement or an arithmetic operation of two variables, it is hard for a model to generate a meaningful test case. To help the model generate meaningful unit tests, we append the target line as extra context/information. Context extraction provides three different outputs. The first output is a set of files identical to the files seen in the project source code structure, but in each file, instead of the full implementation, we only include the method names in that file. The second output is the same set of files, but it includes the method bodies. We also save each method's initial and last line numbers in each file. The third output is all the other parts of the context for each class, consisting of the class name, signatures of the constructor methods, public variables and fields, all other method names, etc. This context design strictly follows the baseline work from the *Methods2Test* dataset, which we used for fine-tuning downstream tasks. After extracting the three context outputs, we iterate over the **Line2test mapping** and concatenate each line with its corresponding focal method and context.

An example of the input of the dataset is demonstrated in Figure 2. When lines are mapped to tests, we can end up with multiple test cases covering the same lines. We don't include all the covering test cases because the input data is the same, and the model will have difficulty optimizing if we provide different outputs for identical inputs. So, we select one test per line. We use the naming convention as a typical solution in the literature to map unit tests to source code. We have the test name and path in our **line2test mapping** and the class name to which the input line belongs. So, we search for the class name of the input line in different tests that have covered the line; if we have a match, we will select that test as the covering test for the input line. If no tests have the same class name as the input line, we include the first test in the list as the unit test. We



```
[LINE] return this.labelInsets; [LINE]
public RectangleInsets getLabelInsets() { [EOL]
            return this.labelInsets; [EOL]
        }
protected Axis(String label);
public boolean isVisible();
public String getLabel();
public void setLabel(String label);
public Font getLabelFont();
public void setLabelFont(Font font);
public Paint getLabelPaint();
public void setLabelPaint(Paint paint);
public RectangleInsets getLabelInsets();
public void setLabelInsets(RectangleInsets insets);
public void setLabelInsets(RectangleInsets insets, boolean notify);
public String getLabelURL();
boolean DEFAULT_TICK_MARKS_VISIBLE=Optional[true];
Stroke DEFAULT_TICK_MARK_STROKE=Optional[new BasicStroke(1)];
Paint DEFAULT_TICK_MARK_PAINT=Optional[Color.gray];
float DEFAULT_TICK_MARK_OUTSIDE_LENGTH=Optional[2.0f];
boolean visible;
String label;
Font labelFont;
RectangleInsets labelInsets;
double labelAngle;
String labelToolTip;
String labelURL;
float minorTickMarkOutsideLength;
Paint tickMarkPaint;
double fixedDimension;
```

Target line
Focal method (FM)
Focal class constructor (C)
Signatures of the other methods in the focal class (M)
Public fields of the focal class (F)

Figure 2: An example method and its context.

select the first match as the mapped unit test if there are multiple matches.

After creating the dataset consisting of the input line, the context on the input side, and the corresponding test that covers the line, we use them to apply *domain adaptation* to the fine-tuned model. The *domain adaptation* enables the model to adapt to the new domain (project) and generate more accurate tests with a higher compilation ratio. The domain adaptation technique we have used is training the model for a few epochs on the generated dataset (much less than a full training). Other options could be freezing the model and adding extra layers (which we did not see a significant benefit in performance from our preliminary experiment). However, this part of our approach can be easily replaced with other techniques in future work.

Note that the computation cost of domain adaptation per project is affordable since it is a one-time cost. If the project tends to change over time, we can re-train the model per sprint. Generating inferences requires significantly less time than classic test case generation approaches, i.e., SBST tools. However, it is worth mentioning that SBST approaches do not require this non-trivial training time. So, deciding which approach is more time-effective is worth investigating for future work.

4 EXPERIMENT SETTINGS

4.1 Research Questions

This study aims to evaluate the effectiveness of code models in generating unit test cases. In addition, we also study the effect of project-level domain adaptation. To address these objectives, we design and study the following three Research Questions (RQs):

4.1.1 ***RQ1 (Performance)****.* How effectively does *CodeT5* generate unit test cases without domain adaptation?

Motivation: Code models such as *CodeBERT* and *CodeT5* have been successfully applied to automated software engineering tasks such as comment generation, defect prediction, and program repair. However, not many studies apply them to test case generation tasks. The few existing studies, such as [4, 40], mainly rely on the static evaluation of the generated test cases using metrics such as BLEU. The problem with this approach is that the generated test may be (a) uncompilable and (b) not effective since they are measured by execution. Therefore, we replicate a state-of-the-art study in this domain and adequately evaluate them by running test cases and calculating their test adequacy metrics. We take *AthenaTest* [39] as our baseline. However, they have not published their model publicly, and our attempt to access it privately was also unsuccessful (due to confidentiality). Also, they have not reported the test adequacy metrics (e.g., line coverage and mutation score). So we replicated their work the best we could, using a similar model *CodeT5* [41] and evaluated it on our dataset.

4.1.2 ***RQ2 (Effectiveness)****.* How effective is project-level domain adaptation in improving the generation quality of test cases using *CodeT5*?

Motivation: As we show in Section 5.1, the results of RQ1 are not promising. One potential explanation is the inability of the models to learn project-specific patterns or knowledge. It is a typical problem in most software engineering tasks where "domain shift" is extreme when a trained model is applied on a brand new project [46]. In most well-maintained real-world projects, the code base already includes developer-written test cases that may cover most of the code. In RQ2, we propose to leverage this data to adapt the domain in RQ1 results to each project. In this RQ, we compare the results to other baselines, namely, *GPT-4* and *A3Test*. We added *GPT-4* as a method to represent the current state-of-the-art LLM and *A3Test* as the state-of-the-art technique specially designed for unit test case generation exploiting domain adaptation from test oracle generation task.

4.1.3 ***RQ3 (Impact)****.* Can our proposed test generation approach augment search-based test case generation?

Motivation: Given that search-based test case generation approaches are state-of-the-art techniques for test case generation, in RQ3, we compare our approach with *EvoSuite* as a well-known and influential search-base approach[12]. One of the motivations for using our approach compared to a technique like *EvoSuite* is the speed of test generation. Given that test generation in our approach only requires one single "inference", the run-time per test case should be substantially lower than search-based baselines. On the other hand, those techniques do not need a lengthy (one-time) training phase beforehand. Our objective for the proposed test generation framework is not to replace search-based approaches but to complement them. We argue that search-based approaches and our Transformer-based methods can target different types of tests to generate. Therefore, using both tools together will be most beneficial. In particular, we envision applying *EvoSuite* once per class to reach a certain level of coverage. Then, running our models (domain adapted on the developer-written tests), we can generate extra test cases covering some new lines on top of the coverage achieved by *EvoSuite* and developer-written tests combined. After the initial round of test generation, our model can be run after each small commit to cover those few new source code lines. We will discuss the complementary nature of the two approaches in Section 5.3. We also compare the mutation score of both methods and report how many new mutants our method can kill compared to *EvoSuite*.

4.2 Datasets

We use two datasets in this study namely *Methods2Test* and *Defects4j*. The *Methods2Test* dataset is used to fine-tune our model for test generation downstream tasks. *Defects4j* projects are then used to adapt the fine-tuned model at a project level and test their final output per project. In the following, we provide more details about each of these datasets.

Methods2Test [39] is the "test generation data" in Figure 1, which consists of Java methods mapped to their corresponding focal methods. It was built on data from 9,410 unique repositories (91,385 original repositories analyzed). The dataset consists of 780,944 instances, divided into training (80%), validation (10%), and test (10%) sets. Since running all the tests, getting the coverage, and mapping the tests to the methods they cover require a lot of manual work and execution time, they have used the naming convention as a heuristic to map each test to its focal method. They search for method names in the test case and map the method mentioned to the test that calls it. They provide the context data as the following: the focal method (FM), focal class name (FC), signatures of the constructor methods of the focal class (C), signatures of other public methods in the focal class (M), and public fields of the focal class (F) – FM+FC+C+M+F. The reason behind these additional contexts is to provide enough information for the model to generate meaningful and compilable tests. For example, a test case may need to instantiate an object of the focal class. The model requires the proper context to generate a statement to instantiate them. Providing the constructor's signature and the focal class's class methods helps the model generate correct instantiating code that does not throw compilation errors. Based on their results, this combination of contexts has shown the best results. We have used all the data provided per method to fine-tune our model for test case generation tasks.

Defects4j [16] is a dataset of Java projects consisting of a collection of reproducible bugs and a supporting infrastructure to advance software engineering research. The initial version of *Defects4J* contains 357 real bugs from 5 real-world open-source projects. Each project also has a comprehensive test suite that can expose each bug in each version of the project. Each version can be accessed using the provided scripts to check out different commits. The updated version of *Defects4j* has 17 projects. In our experiments, we used 5 out of the 17 projects. Section 5 provides more details on the selection criteria of projects.

4.3 Pre-Trained Code Model

In our study, we have chosen *CodeT5* as our code model for the following reasons:

Firstly, we did not choose the recent LLMs, e.g., *GPT-4*, *PaLM*, and *LLaMA*, as they are too large to cost-effectively fine-tune downstream tasks and adapt the domain for the project data. Even though *CodeT5* is much smaller than the recent models, its size gave us a reasonable cost while beating the large LLMs, i.e., *GPT-4*.

Secondly, out of the smaller models, we have chosen *CodeT5* as it is based on *T5* [28]. *T5* is a Transformer that uses denoising sequence-to-sequence (Seq2Seq) pre-training and has shown promising results for understanding and generation tasks. *T5*-based models are better for our use case than *BERT* or *GPT*-based models, given that they are not encoder-decoder models. Despite their success, *BERT*-based models are encoder-only, and *GPT*-based models are decoder-only architectures. Encoder models are usually best suited for understanding tasks, and decoder models are suitable for generation tasks. However, the test case generation task requires both understanding and generation skills. For instance, prior work that leverages *BERT* for a generation task (code summarization) [10] had to add a separate decoder, which does not benefit from pre-training.

Lastly, *CodeT5* considers the token type information in the code data. Most other code-aware models use the conventional NLP pre-training techniques. However, code data has rich structural information essential for fully understanding code functionality.

Therefore, given the resources and the application we needed to train our method, we used *CodeT5* as our code model.

CodeT5 has two main configurations: *CodeT5-small* with 60M parameters and *CodeT5-base* with 220M parameters. The maximum source and target sequence lengths are 512 and 256, respectively. There are different downstream tasks that *CodeT5* can be fine-tuned on, such as summarize, *CONCODE* (text-to-code generation), translate(Code-to-code translation), code refinement, code defect detection, and code clone detection. Two of the tasks mentioned above can be used for test generation, which consists of code and test as input. Based on our initial experimental results, the *CONCODE* task generated better test cases than the code-translation task. It seems rational as the code translation model generates code in a different programming language with the same semantics as the original code. However, the input and output do not have the same structure and semantics in test generation. Therefore, in our experiments, we have selected the *CONCODE* sub-task for training the *CodeT5* model with the *CodeT5-base* (the larger) configuration. We have used the default hyperparameters for the *CodeT5-base* model. The only change we made in the configuration was the batch size due to the limited GPU memory. We used four as our batch size, the batch data that fit most in our GPU memory.

4.4 Baselines

We chose the following two baselines to compare our results to other methods.

(1) ***GPT-4*** is a highly advanced LLM that can mimic human-like speech and reasoning by training on a vast library of existing human communication. It can solve complex problems more accurately, generate creative content, and exhibit human-level performance on various professional and academic benchmarks.
(2) ***A3Test*** is a test case generation method that exploits domain adaptation at a task level, i.e., test oracle generation, in which they transfer the knowledge learned from oracle generation to a whole unit test case generation.

By adding these two methods as our baseline, we aim to assess how our project-level domain adaptation works compared to the state-of-the-art LLM and a novel test case generation method that exploits a task-level domain adaptation.

4.5 Evaluation Metrics

In this paper, we use seven different evaluation criteria to evaluate the performance of test generation, i.e., parse rate, execution rate, line-level code coverage, mutation score, adapted mutation score, BLEU score, and CodeBLEU score.

4.5.1 Parse and Compile Rate. We report the ratio of parsable and compilable test cases generated by each baseline. We use Tree-sitter [2] parser to evaluate the syntax correctness of the generated test case. Within the parsable test cases, we inject them and compile the project to assess the compilability of the generated test cases.

4.5.2 Line-Coverage and Mutation Score. We use line coverage and mutation score for the test case adequacy metric. From a practical point of view, a useful metric for evaluating a test case is a test adequacy metric such as code coverage or mutation score. Other metrics are provided for comparison with baseline literature and their simplicity. Although most studies use BLEU score [25] or CodeBLEU [29] for evaluating the quality of generated code or test case, these metrics are sub-optimal for evaluating testing efficacy. For instance, both BLEU and CodeBLEU are calculating the similarity of the generated output with the ground truth. So, different identifier tokens in the generation will significantly affect its scores while not affecting the functionality of the code or test case. The model can generate a compilable, meaningful, and effective test case that is not similar to the ground truth. Therefore, we picked a simple and basic coverage metric (line coverage) and standard mutation score as our evaluation metrics. In the future, the study can be extended with other adequacy metrics.

To calculate line coverage, we select only the compilable generated test cases, inject them into the project, and run them. For RQ1 and RQ2, we calculate the line coverage of the total project, excluding the lines in the test project, i.e., src/main/test. For RQ3, we calculate the exact code line from the input the generated test case covers. Clover [1] calculates the exact mapping between each test case and its covered lines in the code. To use Clover, we need to add instrumentation scripts to the build system of each project under study. Since different projects in *Defects4j* may use other build systems, we included 5 out of 17 projects compatible with Clover.

To emphasize the practical usefulness of our work, we also report mutation score, a fault-based adequacy metric, to demonstrate that our approach can find bugs that the developer-written test, the most related work, and the search-based baseline method can not. Using their defined mutation operators, we report the standard mutation score (the number of killed mutants divided by the total number of mutants) reported by the Major mutation testing tool. For RQ3, we also report (when applicable) the adapted mutation score, which is the number of killed mutants divided by the covered mutants.

4.5.3 BLEU and CodeBLEU. These two metrics calculate the similarity of the generated code compared to the ground truth. Since integrating test adequacy metrics in the training loop is not feasible (it requires execution, which is very costly), we used BLEU to select the best model during the training. We also provide the BLEU and CodeBLEU scores to compare the model before and after domain adaptation. BLEU calculates the similarity of two texts by calculating their N-Gram co-occurrence. Since BLEU only evaluates the textual similarity, it is not considered optimal in calculating the similarity of two code snippets. While BLEU solely calculates the co-occurring n-grams of tokens, CodeBLEU leverages a weighted n-gram to encapsulate different importance of keywords (i.e., *public*, *int*, *return*). It uses syntactic matching via AST and semantic matching via data-flow [15].

4.6 Configurations and Environment Setup

We have used the *CONCODE* downstream task with *CodeT5*-base configuration as our model. We fine-tune the model on the *Methods2Test* dataset and evaluate the results using *Defects4j* projects and *EvoSuite*-generated tests. All the model hyperparameters are set as the default for the *CONCODE* configuration of *CodeT5*. The batch size is set to 4, the maximum that can fit our setup's GPU memory. We fine-tune all the model layers for 20 epochs for the domain adaptation step.

We have used a single ComputeCanada (Beluga) node for all experiments, with 4 32GB V100 GPUs, 10 CPU cores, and 80GB RAM. However, with minor changes in batch size and training time, all experiments can be executed with 16GB GPU, 1 CPU core, and 10 GB RAM.

5 RESULTS AND ANALYSIS

5.1 RQ1: Effectiveness of *CodeT5* without DA

Experiment Design. First, we explain how we split *Defects4j* data into training and evaluation sets per project. In RQ1, we only use the evaluation set to assess the base model *CodeT5* without Domain Adaptation (DA).

There are two ways to split the train and evaluation set on *Defects4j*. We can randomly select 20% of the lines and move the line-test tuples to the evaluation set. The problem with this approach is that we may have a data leak between the train and the evaluation set. For example, in a method with five lines, each line is mapped to a test. In some cases, all five lines are mapped to the same test case. If we randomly pick 2 of 5 lines for the evaluation set, the model will have access to the other three lines, which consist of the same output test that we expect the model to generate in the evaluation. Since this constitutes a data leak, we divided the data at the test case level.

In this approach, a leave-one-out evaluation [18], we first make a set of all unique test cases in the dataset per project. We randomly select 20% of them for the evaluation set per project. Finally, the evaluation set is created using the line-test tuples of those 20% test cases. This way, the dataset will not have identical test cases between the evaluation and training sets, i.e., no data leaks. Note that some test cases might still cover some lines in the evaluation set in the training test. However, those test cases in the training set are not the same as the main test cases in the test set, which was selected in the 1-to-1 mapping procedure for that given line, which the model tries to generate. We do not consider the following a data leak since the target test cases generated will no longer be the same as those seen in the training.

[2] https://tree-sitter.github.io/tree-sitter/

Table 1: Evaluation metrics scores for *CodeT5* without Domain Adaptation (DA) (RQ1). Comparison of *Code-T5* with DA) versus *GPT-4* and *A3Test* is also shown for RQ2. Bold values denote the best metric score for each project compared to the baselines.

Baselines	Metrics	compress	gson	jksnCore	jksnDB	jsoup	AVG
CodeT5 without DA	Parse Rate	20.75	24.01	14.21	18.26	39.27	23.30
	Compile Rate	1.66	3.67	0.70	0.92	22.51	5.89
	BLEU	11.59	18.64	16.39	18.34	25.56	18.10
	CodeBLEU	9.15	16.64	16.98	16.78	22.10	16.33
	Line Coverage	2.00	25.60	2.10	31.40	63.10	24.84
	Mutation Score	0.55	12.26	0.07	11.95	32.78	11.52
CodeT5 with DA	Parse Rate	89.29	**100.00**	93.33	94.46	**100.00**	95.42
	Compile Rate	**39.29**	**47.67**	**38.33**	**28.37**	**62.50**	**43.23**
	BLEU	**40.84**	**42.06**	**28.41**	**36.74**	**44.36**	**38.48**
	CodeBLEU	**22.37**	**35.12**	**30.06**	**31.70**	**44.10**	**32.67**
	Line Coverage	**32.80**	**52.20**	**21.20**	**43.10**	**68.00**	**43.46**
	Mutation Score	**20.53**	**35.61**	**8.70**	**28.60**	**46.42**	**27.97**
GPT-4	Parse Rate	**99.28**	98.55	**98.37**	**98.08**	99.40	**98.74**
	Compile Rate	2.90	17.15	4.20	7.52	22.75	10.90
	BLEU	18.53	26.39	18.29	22.43	27.11	22.55
	CodeBLEU	18.73	28.19	23.32	23.87	25.65	23.95
	Line Coverage	0.70	32.40	4.10	33.20	56.80	25.44
	Mutation Score	0.10	15.93	0.98	14.44	43.45	14.98
A3Test	Parse Rate	53.70	64.61	44.98	68.16	56.25	57.54
	Compile Rate	1.93	6.85	1.27	1.07	17.14	5.65
	BLEU	11.33	16.44	13.08	15.75	18.79	15.08
	CodeBLEU	7.42	15.61	13.32	15.58	18.11	14.01
	Line Coverage	2.00	29.50	2.00	31.60	52.80	23.58
	Mutation Score	0.00	12.85	0.01	11.95	34.98	11.96

For RQ1, we investigate the ability of *Code-T5* to generate test cases by only applying fine-tuning on downstream tasks. We use the *Methods2test* dataset to fine-tune the test generation task. After training the model on the *Methods2test* dataset, we directly evaluate the model on the evaluation set split, as mentioned above for splitting *Defects4j*. To generate test cases for *Defects4j* projects, we need to extract a context similar to the structure of the *Methods2test* dataset. After generating the tests, we calculate the line coverage on each project. The line coverage is the number of covered lines divided by the total number of lines in the src/main/java folder. The line coverage on the *CodeT5* without DA baselines shows the line coverage by the tests generated by the model without domain adaptation on the evaluation project. We only use test cases that pass for calculating the mutation scores, as we need a green test suite to set up the mutation testing process.

Results. As demonstrated in Table 1, we calculate the evaluation metrics of model-generated tests on five Defects4j projects using *CodeT5* without DA.

We have two findings in this RQ: (a) In most cases, the model-generated test cases were not compilable due to the inability to infer the correct dependencies, which led to the generation of undefined objects. Also, the model generated truncated test cases to the limited output length per sample in *CodeT5* (512 tokens). (b) Existing studies such as [39] only reported the coverage (a high coverage in this case for a small (18) set of sample methods), which is not representative of the actual quality of the model. Most other studies only report generic static metrics, such as the BLEU score, which fails to capture the test adequacy. However, our results revealed that the generated test cases' test adequacy metrics (line coverage and mutation scores) are not as promising as the BLEU or CodeBLEU. They are also much lower than the reported coverage in the original *AthenaTest* paper for the 18 small sample codes they have assessed.

Answer to RQ1: The results of *CodeT5* without DA show that fine-tuning with only task-specific data is insufficient to generate test cases that are compilable or test-adequate.

5.2 RQ2: Effectiveness of CodeT5 with DA

Experiment Design. In this subsection, we explain the details of the experiment procedure used in RQ2.

We use the same splits of Defects4j as mentioned in the previous RQ. To apply domain adaptation, we use the training set of Defects4j to train the fine-tuned model, i.e., *CodeT5* with DA. Then, we generate test cases using the same evaluation set to calculate the metric scores.

We have two state-of-the-art baselines to compare our approach: *GPT-4* and *A3Test*. To generate test cases by *GPT-4*, we had to develop a new style of feeding the input, as our dataset has a lot of

```
prompt = [
    {"role": "system",
     "content": f"You are a unit test case generator
      with meaningful assertions for Java project: {prj}."},
    {"role": "user", "content": f"""Given a focal method
      surrounded by ???, generate unit test case methods
      that cover maximum line coverage. Only create new
      tests if they cover new lines of code. Only generate
      the Java code part of test methods. Use [TCS] to
      separate the multiple test cases. Input text:
      ???{method}???"""},
    {"role": "user", "content": """Remove all comments
      (e.g. line starts with // and surrounded by /* and */),
      NL description and @Test annotations. New lines
      should be substituted with [EOL]."""}
]
```

Figure 3: Prompt used for GPT-4

redundancy due to its granularity being at the line level. Asking the model to create tests for each line individually wastes resources. To mitigate this issue and optimize prompting cost (to make this solution more practical), we ask *GPT-4* to create as many tests as it needs to maximize line coverage for a given method.

The prompt template is shown in Listing 3. First, we query the model with a system prompt that defines the model's role, a unit test case generator with meaningful assertions (a nontrivial requirement for generating unit test cases [42]). For the actual task, we provide the focal method and its context and ask the model to generate a unit test case that covers the maximum line coverage for the focal method. We let the model generate as many tests as it needs (since other baselines of comparisons create multiple tests as well), but to avoid redundant tests, we ask to generate new tests only if they cover new lines of code. We also added minor instructions to the prompt to make our post-processing easier, i.e., only generating Java code, using [TCS] tokens to separate multiple test cases, removing natural language comments and @Test annotations, and substituting new lines with [EOL] tokens.

For *A3Test*, we use their already fine-tuned model and their testing script provided in their replication package. Since *A3Test* is also a token-to-token generation model that receives a structure similar to ours in the input, i.e., focal method + focal context. We use the same evaluation set splits from our Defects4j dataset (from RQ1). All the hyper-parameter settings were used according to what was reported in their paper or the default values suggested in their replication package.

Results. Table 1 reports the evaluation metrics to compare *CodeT5* with DA and the studied baselines. *CodeT5* without DA refers to the *CodeT5* model fine-tuned on test generation downstream task using the *Methods2test* dataset, without any domain adaptation. *CodeT5* with DA refers to the model after applying project-level domain adaptation. The results show that using project-specific data for domain adaptation significantly increases the model's performance. The average improvement of percentage points over all projects is 72.12% for parse rate, 37.34% for compile rate, 20.38% for BLEU, and 16.34% for CodeBLEU, 18.62% for line coverage, and 16.45% for mutation score.

The results of test adequacy metrics are new and promising. As discussed, most related work does not report these metrics due to the effort required to make all test cases executable. Our results reveal that without project-specific domain adaptation, the metrics are low, with line coverage between 2% to 63%, with a median of 25.6% and a mean of 24.84%. For mutation score, it ranged between 0.07% to 32.78%, with a median of 11.95% and a mean of 11.52%. However, the metrics improve significantly after applying the domain adaptation, with line coverage to a range between 21.20% and 68%, a median of 43.10%, and a mean of 43.46%. For the improved mutation score, it ranged between 8.70% and 46.42%, with a median of 28.60% and a mean of 27.97%. In other words, there was a 17.5% improvement over the median and an 18.62% improvement over the mean for line coverage; 16.65% improvement over the median, and 16.45% over the mean for mutation score in percentage points. From the result, we can observe that applying domain adaptation to transfer project-specific knowledge has a substantial improvement in unit test generation, both in static textual similarity and test adequacy metrics. The reason could be that unit test generation heavily relies on the internal knowledge of the software under test. Just tuning the models at a task level without enough knowledge of the software will have a marginal effect on the testability of the generated unit tests.

We also compare our approach with two state-of-the-art baselines, i.e., *GPT-4* and *A3Test*. As shown in Table 1, none of the baselines could outperform *CodeT5* with DA in all metrics except for the parse rate of *GPT-4*. *GPT-4* showed better overall performance than *A3Test* in all metrics. *A3Test* and *CodeT5* without DA showed the least performance. *A3Test* had the lowest performance in compile rate, line coverage, BLEU, and CodeBLEU. *CodeT5* without DA showed the least parse rate and mutation score performance. The performance difference between the two least-performing baselines was not very big. However, the difference between their parse rate scores was significant, with +34.24% points for *A3Test*. The results suggest that transferring Oracle generation knowledge has a positive impact in generating syntactically correct test cases.

One interesting observation is that CodeBLEU scores are relatively higher than BLEU for *GPT-4*. Even though it generates different n-gram tokens than the ground truth, the AST-matching and dataflow matching scores are relatively higher. Even though *GPT-4* generates different tokens, e.g., different identifier names, it generates similar code in terms of syntax (AST) and semantics (dataflow). Generating different identifier tokens can be an inherent trait of *GPT-4* as it is trained on a much more diverse dataset with much larger parameters. However, there is more than one way of naming an identifier in source code. What determines the function of a code is its syntax and semantics. Also, *GPT-4* shows the best performance in parse rate, meaning that it generates the most syntactically correct test cases. However, they could not beat *CodeT5* with DA in other metrics as they lack the domain-specific knowledge to make the code locally correct for compilation and effective test adequacy. From this observation, we foresee good potential on *GPT-4* if paired with the proper tuning strategy, e.g., prompt-tuning, fine-tuning, domain adaptation, etc.

Table 2: Line Coverage comparison between *EvoSuite* and model generated tests. The NewCL has covered lines that neither *EvoSuite* nor the developer-written tests from the training set have covered.

Project	Model CL		EvoSuite CL		New CL		Total Lines
compress	216	58%	87	23%	174	46.70%	372
gson	458	69%	539	82%	31	4.70%	657
jksnCore	399	30%	674	51%	82	6.20%	1307
jksnDB	1357	50%	136	5%	1246	48%	2595
jsoup	192	82%	39	16%	157	66.50%	519
AVG	524.4	58%	295	35%	338	34.42%	1090

Table 3: Mutation and adapted mutation scores for model generated (Model MS and Model AMS) and *EvoSuite* (Evo MS and Evo AMS) tests. The New MK column shows the number of mutants not killed by *EvoSuite* but by model-generated tests.

Project	Model MS	Evo MS	Model AMS	Evo AMS	New MK	
compress	0.00%	55.90%	0.00%	69.50%	0	0%
gson	13.50%	64.90%	50.00%	100.00%	0	0%
jksnCore	14.80%	87.20%	50.70%	100.00%	0	0%
jksnDB	22.40%	0.00%	54.20%	0.00%	26	22.40%
jsoup	32.00%	0.00%	47.10%	0.00%	8	32%
AVG	16.54%	41.60%	40.40%	53.90%	6.8	11%

Answer to RQ2: Overall, the results suggest that applying project-specific domain adaptation improves *CodeT5* by **72.12%** in parse rate, **37.34%** in compile rate, **20.38%** in BLEU, **16.34%** in CodeBLEU, **18.62** in line coverage, and **16.45%** in mutation score over the one without DA. It also significantly outperforms all the other baselines, except for the parse rate of *GPT-4*.

5.3 RQ3: Augmentation with SBST

Experiment Design. In RQ3, we compare our approach with *EvoSuite*, a well-known search-based approach for test case generation. First, we run *EvoSuite* with the default settings (10 minutes per class) to generate tests for all classes in the project. We use the same train-test split as RQ1 and RQ2. We calculate the line coverage and the mutation scores using the *EvoSuite* framework. The purpose of our proposed framework is not to compete with or replace *EvoSuite* but to complement or augment such existing approaches. Therefore, we also report the number of "new lines" our test cases can cover compared to what was covered already by the *EvoSuite*-generated test suite. Also, note that these "new lines" are not covered by the developer-written test cases of the training sets either. Thus, the study emphasizes the tool's impact by comparing existing automated testing tools and manual test generation practices.

Results. Table 2 reports the line coverage of the test cases generated by *EvoSuite* and our framework. The model-covered lines (Model CL) column shows the number and percentages of lines of code covered by model-generated tests. *EvoSuite*-covered lines (*EvoSuite* CL) show the number and percentages of lines covered by *EvoSuite*. Finally, the new covered lines (New CL) column shows the extra lines covered by model-generated tests that *EvoSuite* can not cover. Note that these lines are not covered by the developer-written test cases of the training sets either.

The results indicate that *EvoSuite* line coverage is higher than our framework in 2 projects, and ours is higher in 3 projects. Overall, *EvoSuite*'s median and mean coverage are 23% and 35.4% vs. ours, which are 58% and 57.8%. However, the motivation of our work is to augment existing test generation systems and not to replace them. If our model generates tests that can cover new uncovered lines compared to *EvoSuite*, we say our model augments *EvoSuite*; the total coverage will be more than both individually. Looking at the New CL column, we see that in 5 out of 5 projects, we can augment *EvoSuite* by covering extra lines.

Table 3 reports the mutation score of model-generated unit tests compared to the *EvoSuite* tests. We used defects4j to calculate the mutation score. The low mutant coverage of our approach is because we are using only 20 percent of each project as our test set, but mutants are everywhere. Since the data is divided in a line-level manner and *EvoSuite* needs the whole class for test generation, we could not use *EvoSuite* to only generate tests for the test set portion of the dataset (the portion that was given to the trained model for test generation). So, a direct comparison is not straightforward.

The same problem also exists in the coverage calculation since *EvoSuite* generates a test suite for the whole project. However, we could select only the lines in our test set for coverage comparison using Clover coverage reports, which we cannot do with the mutation tool. To better reflect the mutation-killing power of our approach, we calculated the Adapted Mutation Score, which compares the model's ability to kill the covered mutants.

Comparing the two techniques, the mutation score of model-generated tests is higher than *EvoSuite* for 2 (Jacksondatabind, Jsoup) out of 5 projects. For example, in jksnDB, we kill 26 new mutants.

As mentioned, all the above results for *EvoSuite* are collected with the default *EvoSuite* setup, which is 10 minutes timeout per class. One can argue that *EvoSuite* might generate more new lines if we set a higher time. Although this is true in theory, first, the default values are chosen based on hyper-parameter tuning, which means that, on average, one won't get much more coverage by simply giving more time for test generation per class. Second, we also noticed that most projects would converge even before 10 minutes.

As explained before, we recommend using our approach in addition to a tool like *EvoSuite*. Our suggested use case in practice is to start with *EvoSuite* (with a default budget). Then, identify lines not covered by *EvoSuite* and pass them to our framework to generate test cases instantly. Note that one test case generation in our framework takes around 2 seconds (including all post-processing steps) compared to 153 seconds per test case on average for *EvoSuite* on these projects. Our approach provides a fast add-on to *EvoSuite*, especially for new commits, since otherwise, one would need to rerun *EvoSuite* for the whole class with every minor change.

```
public void testHashCode1609() {
    ArcDialFrame f1 = new ArcDialFrame();
    ArcDialFrame f2 = new ArcDialFrame();
    assertTrue(f1.equals(f2));
    int h1 = f1.hashCode();
    int h2 = f2.hashCode();
    assertEquals(h1, h2);
}
```

Figure 4: An example of model-generated tests.

Finally, note that in addition to improving performance and being faster than the alternative search-based approach, the other attraction of our work is to focus on generating readable and more maintainable test cases, given that they are derived based on developer-written test cases rather than predefined templates of search-based approaches. Although we did not study this aspect in detail in this paper and only showed an example 4, our baseline paper [39] provides some evidence based on their user study.

Answer to RQ3: In general, our approach can increase the coverage and mutation score of the existing state-of-the-art test generation techniques such as *EvoSuite* and thus is recommended to be used together with such tools.

6 LIMITATIONS AND THREATS TO VALIDITY

One of the limitations of our approach is that it might depend on developer-written test cases. Although training our model on automatically generated tests is possible, it could hinder the benefits we were targeting, such as better fault detection and readability. Therefore, our approach's use case is to extend existing tests so that new lines are covered, and new faults are detected. However, in practice, this is not a considerable hindrance since, except for newly created projects, most reasonable projects come with some tests already in their regression test suite. Thus, our approach can start with those test suites and improve and augment them. Alternatively, suppose the project does not have any test cases. In that case, the next best option is using automatically generated test cases that are generated by an LLM such as *GPT-4* so that the initial tests are still readable and have a relatively high quality, to begin with.

Regarding construct validity threats and the effectiveness of the metrics, we made sure we went beyond code coverage and looked at the mutation score. New mutants killed by our approach mean potentially new faults can be detected by the model-generated tool compared to what the developers have detected. However, we did not provide a systematic study for the readability of test cases and only showed an example. We neglected this part since the baseline paper [39] already has done a user study and reported the readability as a benefit of model-generated tests.

Regarding the threat to external validity, one closely related study we failed to compare with was *ChatTester*. We couldn't properly run their tool on the five Defects4j projects used in this study, mainly due to their data-pair collection component. The component collected 20 pairs of data instances from one project and no pairs for the other four projects, which was insufficient for comparison. To mitigate this, we compared our work with *GPT-4* to investigate the test case generation performance of state-of-the-art LLM. Since in our comparison, we did not employ advanced prompt engineering strategies such as those used on *ChatTester*, future studies are needed to compare our work with advanced prompt-engineered LLMs for test generation.

Also, we agree that the selected projects can threaten this study's external validity. However, Defects4j is a well-known and widely used dataset with quality unit test cases. Using a limited number of projects is mainly due to the considerable resource cost of calculating the test adequacy metrics. Due to the same reason, some of the previous studies that evaluate test adequacy metrics (i.e., line coverage or mutation score) on test or test oracle generation also experimented with a small number of projects like us, e.g., TOGA [8] and A3Test [4]. Also, it is worth noting that running GPT-4 per each extra project is very costly, hindering the experiments' size.

7 CONCLUSION

This study showed that code models can be fine-tuned on test generation downstream tasks. However, their performance is ineffective on a new project compared to search-based approaches due to domain shift. To mitigate the problem, we proposed a domain adaptation framework that leverages existing developer-written tests. We showed that applying project-level domain adaptation improves the quality of the generated test cases w.r.t. compilability, similar to human-written and test-adequate. Our approach outperforms the largest state-of-the-art LLM, *GPT-4* on all metrics except the parse rate and all metrics for *A3Test*, the deep test case generation method that exploits task-level domain adaption. Finally, we compared our proposed framework with state-of-the-art search-based approaches and showed that our approach could complement and increase line coverage and mutation score. In the future, we will explore other code models and expand the experiment on new datasets. We will also run our user study to evaluate better the generated tests' readability.

ACKNOWLEDGMENTS

This work was partially supported by the NSERC Discovery Grant (RGPIN/04552-2020), and the NSERC and Alberta Innovates Alliance Grant (ALLRP/568643-2021).

REFERENCES

[1] 2023. *OpenClover code coverage platform for Java and Groovy.* https://openclover.org/

[2] 2023. *Tree-sitter is a parser generator tool and an incremental parsing library.* https://tree-sitter.github.io/tree-sitter/

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[4] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).

[5] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. 2008. On the effectiveness of manual and automatic unit test generation. In *2008 The Third International Conference on Software Engineering Advances.* IEEE, 252–257.

[6] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 107–118.

[7] Lee R Dice. 1945. Measures of the amount of ecologic association between species. *Ecology* 26, 3 (1945), 297–302.

[8] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. https://doi.org/10.1145/3510003.3510141

[9] Abolfazl Farahani, Sahar Voghoei, Khaled Rasheed, and Hamid R Arabnia. 2021. A brief review of domain adaptation. *Advances in data science and information engineering: proceedings from ICDATA 2020 and IKE 2020* (2021), 877–894.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[11] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[12] Gordon Fraser and Andrea Arcuri. 2016. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. 33–36.

[13] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019), 312–327.

[14] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[17] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.

[18] Ekrem Kocaguneli and Tim Menzies. 2013. Software effort models should be assessed via leave-one-out validation. *Journal of Systems and Software* 86, 7 (2013), 1879–1890.

[19] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[20] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 643–653.

[21] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[22] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 382–391.

[23] 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.

[24] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 130–141.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[26] Vishal M Patel, Raghuraman Gopalan, Ruonan Li, and Rama Chellappa. 2015. Visual domain adaptation: A survey of recent advances. *IEEE signal processing magazine* 32, 3 (2015), 53–69.

[27] Gustavo HL Pinto and Silvia R Vergilio. 2010. A multi-objective genetic algorithm to test data generation. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, Vol. 1. IEEE, 129–134.

[28] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[29] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[30] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 287–298.

[31] Laurence Saes. 2018. Unit test generation using machine learning. *Universiteit van Amsterdamg* (2018).

[32] 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* (2023).

[33] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2023. Assessing Evaluation Metrics for Neural Test Oracle Generation. *arXiv preprint arXiv:2310.07856* (2023).

[34] Jiho Shin and Jaechang Nam. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555.

[35] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks. *arXiv preprint arXiv:2310.10508* (2023).

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[37] Taffee T Tanimoto. 1958. Elementary mathematical theory of classification and prediction. (1958).

[38] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 299–303.

[39] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv:2009.05617* (2020).

[40] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64.

[41] Yue Wang, Weishi Wang, Shafiq 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[42] 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*. 1398–1409.

[43] Wikipedia Contributors. 2023. Overlap — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Overlap&oldid=1061948530 [Online; accessed 18-January-2023].

[44] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning. ICSE.

[45] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[46] Armin Zirak and Hadi Hemati. 2022. Improving Automated Program Repair with Domain Adaptation. *arXiv preprint arXiv:2212.11414* (2022).

Received 15 December 2023; revised 9 February 2024; accepted 2 March 2024