Evaluating the Robustness of Deep Learning Models on Automated Program Repair

Yu Shi

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

January 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Yu Shi**

Entitled: **Evaluating the Robustness of Deep Learning Models on Automated Program Repair**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Nikolaos Tsantalis

_____ External Examiner
Dr. Weiyi Shang

_____ Examiner
Dr. Nikolaos Tsantalis

_____ Supervisor
Dr. Jinqiu Yang

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

December 19, 2022

_____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Evaluating the Robustness of Deep Learning Models on Automated Program Repair

Yu Shi

Automated Program Repair (APR) helps improve the efficiency of software development and maintenance. In recent years, Deep Learning (DL) approaches have been applied to the APR field and have shown promising potential in fixing software bugs automatically. The DL-based APR models translate buggy code to correct code directly. Some recent works test the general performance of various deep learning models on downstream tasks, e.g., code search and method name prediction. However, there still needs to be a fair evaluation of the deep learning models on automated program repair.

This paper aims to quantitatively and comparatively evaluate the repair performance and robustness of DL-based APR models. We first fine-tune seven pre-trained models and train two models from scratch on the unified dataset for a fair comparison of repair performance. Then, we conduct a robustness evaluation for nine trained models above against nine semantic-preserving code transformations. Our experiments show that DL-based APR models with pre-training perform better repair performance and robustness than those trained from scratch. Additionally, most APR models fine-tuned on the concrete code datasets have better repair performance than those fine-tuned on the abstract code datasets. Furthermore, most encoder-decoder-based and decoder-based APR models have better repair accuracy than encoder-based ones. Finally, compared with renaming-related code transformations, semantic-preserving transformations related to the change of syntactic structure have a more significant impact on the repair robustness of DL-based APR models. The results provide useful insights for achieving better DL-based APR approaches.

Index Terms–automated program repair, deep learning, robustness testing

Acknowledgments

Foremost, I am extremely grateful to my supervisor Dr. Jinqiu Yang. I learned a lot from her, including how to think critically about research, how to shape ideas, how to push forward projects, and how to communicate with other people and put myself in others' shoes. What I learned from Dr. Yang will benefit my future research career.

I sincerely thank my thesis examiners: Dr. Weiyi Shang and Dr. Nikolaos Tsantalis, for spending their valuable time reviewing my thesis and providing constructive suggestions. Their valuable comments make this thesis better.

Furthermore, I thank my friends and lab mates: Bo Yang, Junjie Li, Fazle Rabbi, and Triet Pham, for their help in my difficult times and for happy memories of funs together. I also thank my friends Zishuo Ding, Zehao Wang, Hetong Dai, Yunqi Xu, Yiwen Heng, and Kunyi Wang for their help and funs together. I learned a lot from you.

Last but not least, I would like to express my thanks to my families: my parents, my brothers, and my sister for their continuous encouragements. They unconditionally support me in pursuing my dream. Special thanks to my twin brother, who is always with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automated program repair (APR) has become a hot intersection topic of software engineering and artificial intelligence, which aims to ensure software quality by automatically fixing bugs without human intervention. Over the last decade, the most popular APR approaches are generate-and-validate (G&V) methods (Hua, Zhang, Wang, & Khurshid, 2018; J. Jiang, Xiong, Zhang, Gao, & Chen, 2018; Kim, Nam, Song, & Kim, 2013; Le, Lo, & Le Goues, 2016a; Le Goues, Nguyen, Forrest, & Weimer, 2012; C. Liu, Yang, Tan, & Hafiz, 2013; Long & Rinard, 2015; Saha, Lyu, Yoshida, & Prasad, 2017; Wen, Chen, Wu, Hao, & Cheung, 2018; Xuan et al., 2016; B. Yang & Yang, 2020; J. Yang, Zhikhartsev, Liu, & Tan, 2017). These approaches first continuously generate the candidate patches for a buggy program, then utilize the test suites as the specification to validate each candidate patch until a correct patch meets the expectation. However, there is often a need for test cases in practice, and the quality of available test cases is usually far from the expectation of a repair specification.

In recent years, the advances in deep learning (DL) have allowed researchers to start applying DL to various source code processing tasks, including code completion (Li, Wang, Lyu, & King, 2017), code search (Cambronero, Li, Kim, Sen, & Chandra, 2019), code summarization (Wan et al., 2018), code comment generation (Hu, Li, Xia, Lo, & Jin, 2018), code clone detection (H. Wei & Li, 2017), and logging text generation (Ding, Li, & Shang, 2022). Most of these approaches train the models based on big source code datasets mined from open-source software repositories,

which showed promising performance. Particularly, Neural Machine Translation (NMT) based approaches have been applied in the APR field (Berabi, He, Raychev, & Vechev, 2021; Chakraborty, Ding, Allamanis, & Ray, 2020; Chakraborty & Ray, 2021; Z. Chen et al., 2019; Gupta, Pal, Kanade, & Shevade, 2017; N. Jiang, Lutellier, & Tan, 2021; Lutellier et al., 2020; Tufano et al., 2019; Ye, Martinez, & Monperrus, 2022; Zhu et al., 2021), and this end-to-end approach showed great potential without the test suites specifications. The NMT-based models formulate a code repair process as a translation task from buggy code to correct code, similar to the translation task in the natural language processing (NLP) field (e.g., translation from English to Chinese). The core architecture of these approaches consists of an encoder and a decoder. The encoder treats the input as a sequence of tokens and outputs the intermediate representation. Then, the decoder leverages the output of the encoder to generate the token sequence. We will call all such models as DL-based APR models because these models are trained on multiple source code datasets for fixed bugs automatically.

Nevertheless, deep learning models are known to suffer from non-robust issues and can be easily fooled by small perturbations. Recent works have been conducted on testing the performance of code models. Ramakrishnan et al. (Ramakrishnan et al., 2020) investigate the robustness of source-code models by applying k-transformation on inputs. They utilize a Bi-LSTM seq2seq model and a code2seq model as subject models and evaluate them on code summarization. Rabin et al. (Rabin et al., 2021) test the generalizability of code2vec, code2seq, and GGNN models on predicting method names. Bielik et al. (Bielik & Vechev, 2020) verify that, similar to other domains, neural code models are vulnerable to adversarial attacks, and they find that only some parts of the input program are relevant to the model's prediction. So they abstract the rest of the programs, which makes the adversarial training more effective. Wei et al. (M. Wei, Huang, Yang, Wang, & Wang, 2021) present a coverage-based fuzzing framework to test the robustness and generalizability of code models. Pour et al. (Pour, Li, Ma, & Hemmati, 2021) present a search-based framework for testing the performance of code embedding models. Zhang et al. (H. Zhang et al., 2020) propose Metropolis-Hastings Modifier (MHM) identifier renaming technique to generate adversarial examples for code models. Following this, Yang et al. (Z. Yang, Shi, He, & Lo, 2022) also use variable renaming for adversarial examples generation to attack code models.

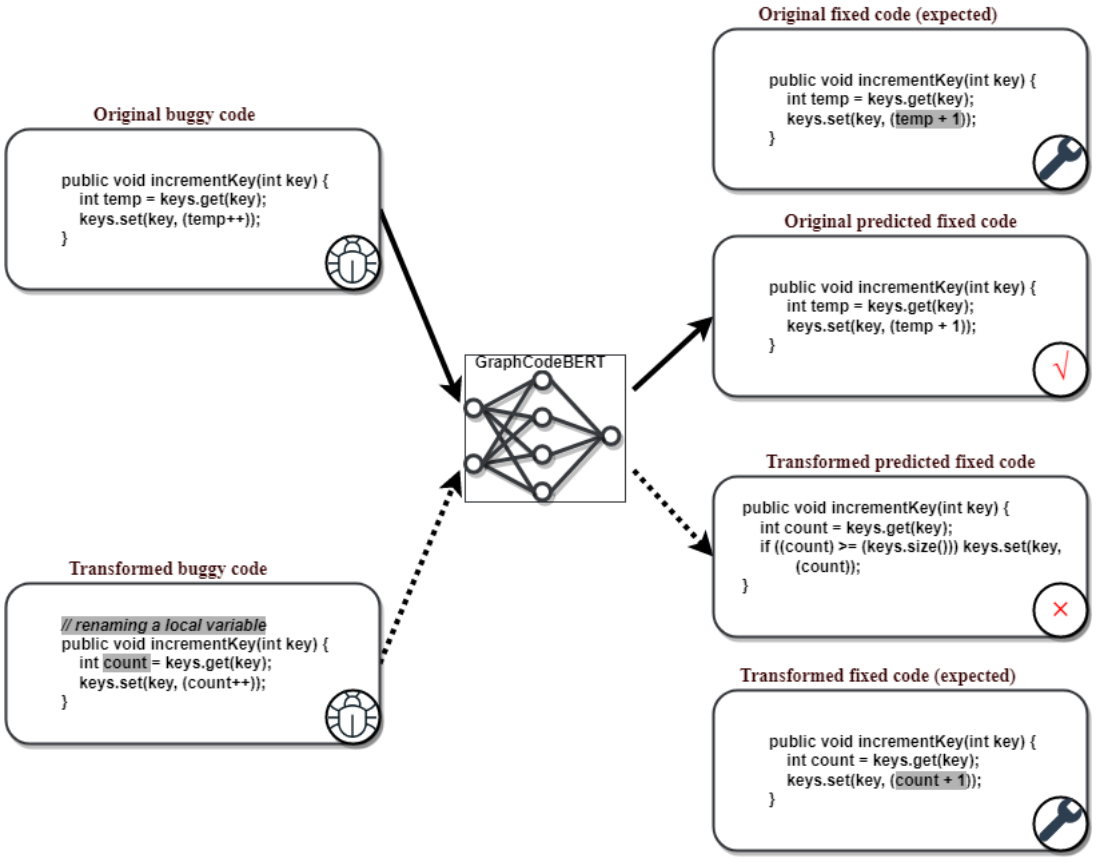

Figure 1.1: An example of program repair of original code and transformed code by GraphCodeBERT (Guo et al., 2020).

However, to the best of our knowledge, there needs to be a fair evaluation of different DL-based APR models regarding repair performance and robustness. Figure 1.1 presents an example of program repair by GraphCodeBERT (Guo et al., 2020). The bug uses an incorrect variable parameter *temp++*, and the expected fixed code uses *temp+1* instead of *temp++*. We first fine-tune GraphCodeBERT on concrete BFPs (Section 3.1 will detail the fine-tuning process), then we feed the buggy code to the fine-tuned GraphCodeBERT. The fixed code predicted by GraphCodeBERT is correct. As mentioned before, deep learning models are known to suffer from robustness problems, and we want to know the predicted fixed code if we apply a slight change to the original buggy code. So we use a substitutive name *count* to replace the original variable *temp*. Finally, we found that GraphCodeBERT predicts an incorrect fixed code for the transformed buggy code. This simple example offers a straightforward observation: Even just renaming a variable's name for the buggy

input code, the predicted fixed code by GraphCodeBERT will change.

This observation intuitively verifies that DL-based APR models may suffer from non-robust issues, as previous research mentioned. We wonder what the things will be for other DL-based APR models, such as whether the CodeT5 (Y. Wang, Wang, Joty, & Hoi, 2021) model can fix this buggy, and if so, whether it will also not fix the transformed buggy code by renaming a local variable. Furthermore, we wonder whether other code-preserving transformations can fool DL-based APR models. If so, which types of transformations can fool the models most? For example, if such models exactly suffer from non-robust issues, what is the distribution of repair performance of DL-based APR models against different code-preserving transformations? In a word, we will investigate the following research questions based on the observations above:

- **RQ1:** What is the repair performance of different DL-based APR models?
- **RQ2:** What is the repair robustness of different DL-based APR models against different semantic-preserving code transformations?

We design two experiments for answering two research questions above. Firstly, we follow the current popular pre-training and fine-tuning paradigm for fine-tuning all subject models on program repair. Note that recent works have started applying large pre-trained models on APR. However, they either use the abstract BFPs dataset proposed by (Tufano et al., 2019) or the corresponding concrete BFPs dataset supplemented by (Chakraborty & Ray, 2021). For example, experiment results in (Lu et al., 2021) show that CodeBERT achieves a good repair result in abstract BFPs, and the following GraphCodeBERT (Guo et al., 2020) achieves a better repair result than CodeBERT because of the additional consideration of code structure information in the pre-training stage. The following PLBART (Ahmad, Chakraborty, Ray, & Chang, 2021) fine-tuned on abstract BFPs and MODIT (Chakraborty & Ray, 2021) fine-tuned on concrete BFPs achieve better repair performance thanks to the encoder-decoder architecture. The latest pre-trained models (e.g., CodeT5 (Y. Wang et al., 2021), SPT-Code (Niu et al., 2022), CoditT5 (J. Zhang, Panthaplackel, Nie, Li, & Gligoric, 2022) and NatGen (Chakraborty, Ahmed, Ding, Devanbu, & Ray, 2022), etc.) report better results on program repair. All of these latest works leverage abstract BFPs as their benchmark dataset. However, no existing works notice that these two different versions of datasets have an impact on

repair results. Here, we design the first experiment for fine-tuning all subject models on unified datasets (i.e., both abstract BFPs and concrete BFPs) for a thorough and fair comparison of the repair performance.

Secondly, we design another experiment that employs the fine-tuned models in the first experiment to evaluate their repair robustness against semantic-preserving code transformations. We first define nine semantic-preserving code transformations, including local variable renaming, method renaming, parameter renaming, boolean exchange, convert switch to if, insert log statement, insert try catch, loop exchange, and reorder condition. Mainly, we apply the same renaming substitution strategy (Z. Yang et al., 2022) for renaming local variables, methods, and parameters. Then, we apply each code transformation on the test dataset of concrete BFPs to generate the applicable transformed dataset and corresponding original dataset. Furthermore, we feed the buggy code of each dataset to models and compare the top-1 predicted fix code with the corresponding expected fix code. Finally, we perform large-scale experiments on the current three types of representative code models, including encoder-based models: CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020); the decoder-based model CodeGPT (Lu et al., 2021); encoder-decoder-based BART-style models: PLBART (Ahmad et al., 2021) and SPT-Code (Niu et al., 2022), and the encoder-decoder-based T5-style models CodeT5 (Y. Wang et al., 2021), including two versions: CodeT5-small and CodeT5-base. We also choose two models trained from scratch (i.e., LSTM-based NMT model and Transformer-based NMT model). We evaluate the results based on two metrics: Accuracy@1 and CodeBLEU (Ren et al., 2020).

Based on our fair evaluation of the repair performance and robustness of different DL-based APR models, we find that: (1). Most encoder-decoder-based APR models and the decoder-based APR model have better repair accuracy than encoder-based APR models. (2). Most DL-based APR models fine-tuned on concrete code datasets have better repair performance than those fine-tuned on abstract code datasets. (3). All subject DL-based APR models with pre-training show better repair accuracy results than those trained from scratch. (4). DL-based APR models exactly suffer from non-robust issues. (5). The semantic-preserving transformations related to the change of syntactic structure have a more significant impact on the repair robustness of DL-based APR models compared with renaming-related transformations. (6). DL-based APR models trained from

scratch show more significant robustness issues than the models with pre-training. Our findings and the lessons we learned from experiments provide important insights for future research.

In a word, the main contributions of this thesis include:

- This thesis conducts a fair evaluation to quantify the repair performance of different DL-based APR models.
- This thesis focuses on comparatively and quantitatively analyzing the robustness of DL-based APR models.
- This is the first work noticing that those DL-based APR models with pre-training show better repair effects on the concrete BFPs dataset than the current mainstream abstract BFPs dataset.
- We provide a summary of the findings and lessons we learned in our experiments. Such insights are valuable for achieving better DL-based APR approaches.

The rest of this paper is organized as follows. Chapter 2 presents the closely related works of this thesis. Chapter 3 presents the setup of our experiments. Chapter 4 details the experiment results and analysis of our research questions. Chapter 5 discusses the threats to validity of this work. Finally, chapter 6 concludes this thesis and presents the potential future works.

Artifacts are available (*APR-Models-Performance*, n.d.).

Chapter 2

Related Work

The works presented here are built on top of three active research areas: automated program repair (APR), pre-trained models, and robustness testing of code models. Specifically, we first present the research on generate-and-validate-based (G&V-based) APR and deep learning-based (DL-based) APR. Secondly, a recent new line of DL-based APR employs large pre-trained models on program repair. So, we briefly present recent pre-trained models and their applications on APR. Finally, we present the recent robustness testing works on large pre-trained models.

2.1 Automated Program Repair

We discuss two areas of related research on APR in this section: G&V-based APR and DL-based APR.

2.1.1 Generate-and-Validate-based Automated Program Repair

Automated program repair has been a well-researched field over the past decade, which aims to automatically fix bugs without human intervention. There are mainly two directions in automated program repair: G&V-based APR approach and synthesis-based APR approach. G&V-based APR approaches (Hua et al., 2018; J. Jiang et al., 2018; Kim et al., 2013; Le et al., 2016a; Le, Lo, & Le Goues, 2016b; Le Goues, Dewey-Vogt, Forrest, & Weimer, 2012; Le Goues, Nguyen, et al., 2012; C. Liu et al., 2013; Long & Rinard, 2015; Mechtaev, Yi, & Roychoudhury, 2015; Qi, Long,

Achour, & Rinard, 2015; Saha et al., 2017; Wen et al., 2018; Xuan et al., 2016; J. Yang et al., 2017; Ye et al., 2022) dominate the APR research and become the most popular methods. Such approaches first continuously generate the candidate patches for a buggy program, then utilize the test suites as the specification to validate each candidate patch until a correct patch meets the expectation. However, the main challenges are that there is often a need for test cases in practice, and the quality of available test cases is often far from expectation as a repair specification. Additionally, identifying the correct patches in a search space takes much time and effort. Specifically, (B. Yang & Yang, 2020) investigates the difference between runtime behaviors modified by plausible patches and correct patches. (Gao, Mechtaev, & Roychoudhury, 2019) proposes an approach to identify correct patches by filtering out over-fitted patches. (J. Yang, Tan, Peyton, & Duer, 2019) proposes an approach to help developers better utilize static application security testing techniques for improving software quality. (Liang et al., 2021) proposes an interactive patch filtering approach to help developers filter out incorrect patches effectively.

2.1.2 Deep Learning-based Automated Program Repair

In recent years, many works started to employ end-to-end deep learning techniques on program repair, which opens up a new APR method: the DL-based APR approach. Most DL-based APR approaches (Ahmad et al., 2021; Berabi et al., 2021; Chakraborty et al., 2022, 2020; Chakraborty & Ray, 2021; Z. Chen et al., 2019; Guo et al., 2020; Gupta et al., 2017; N. Jiang et al., 2021; Lu et al., 2021; Lutellier et al., 2020; Niu et al., 2022; Tufano et al., 2019; Y. Wang et al., 2021; J. Zhang et al., 2022; Zhu et al., 2021) are based on Neural Machine Translation (NMT) architecture. A typical NMT system is formulated as the components of encoder-decoder and attention. These approaches formulate the process of bug repair as a translation mechanism, i.e., translating buggy code to fixed code as translating English to French in natural language processing (NLP). In this field, some works focus on training a repair model from scratch (Chakraborty et al., 2020; Z. Chen et al., 2019; Gupta et al., 2017; Lutellier et al., 2020; Tufano et al., 2019), i.e., they collect a training dataset to train the model with designed training objectives on program repair. Another research line of DL-based APR approaches focuses on leveraging pre-trained models for program repair (Ahmad et al., 2021; Chakraborty et al., 2022; Chakraborty & Ray, 2021; Guo et al., 2020; N. Jiang et al.,

2021; Lu et al., 2021; Niu et al., 2022; Y. Wang et al., 2021; J. Zhang et al., 2022). To be specific, different from the previous approaches that are training the APR models from scratch, these APR models instead are first pre-trained to build the ability of general code understanding and generation with large pre-training datasets. Then, it is further fine-tuned on repair-specific datasets for program repair. Our work mainly focuses on the APR models of the second research line, i.e., we conduct fair and thorough experiments to evaluate the repair performance and robustness of DL-based APR models. The following section further details the recent works on this line of APR research.

2.2 Pre-trained Models

We discuss the related research of pre-trained models in two aspects: pre-trained models on Software Engineering (SE) and pre-trained models on APR.

2.2.1 Pre-trained Models on Software Engineering

In recent years, three types of large pre-trained models have been proposed: encoder-based models (e.g., BERT (Devlin, Chang, Lee, & Toutanova, 2018)), decoder-based models (e.g., GPT-2 (Radford et al., 2019)), and encoder-decoder-based models (e.g., T5 (Raffel et al., 2020) and BART (Lewis et al., 2019)), and they achieve great success in a variety of NLP tasks. These pre-trained models are mainly built on Transformer (Vaswani et al., 2017). They are commonly first pre-trained on large text corpora by various self-supervised training objectives to build the capacity of either sequence understanding or sequence generation, or both. Then the pre-trained models can be fine-tuned on a wide range of task-specific datasets and applied to corresponding downstream tasks. This NLP paradigm has also been utilized to solve various problems in programming language processing (PLP) (Ahmad et al., 2021; Chakraborty et al., 2022; M. Chen et al., 2021; Feng et al., 2020; Guo et al., 2020; Lu et al., 2021; Niu et al., 2022; D. Wang et al., 2022; Y. Wang et al., 2021; Z. Zhang, Zhang, Shen, & Gu, 2022) (e.g., code generation, code search, code translation, and clone detection).

The three types of pre-trained models have all been adapted to PLP and outperformed the previous state-of-the-art approaches. The encoder-based code models (Feng et al., 2020; Guo et al., 2020)

are mainly based on BERT, a bidirectional Transformer-based encoder, which is better at sequence understanding tasks (e.g., code search). An additional decoder is needed for encoder-based models when performing generation tasks. Generally, this decoder is trained from scratch in task-specific fine-tuning stages. The decoder-based models (Lu et al., 2021) are mainly based on GPT-2, which is more suitable for sequence generation tasks (e.g., code generation). However, it can still be applied in code understanding tasks even though it may be a sub-optimal option. The encoder-decoder-based code models work on both code understanding and generation tasks. There are mainly two branches: one branch (Ahmad et al., 2021; Chakraborty & Ray, 2021; Niu et al., 2022) is based on BART, and the other branch (Chakraborty et al., 2022; Y. Wang et al., 2021; J. Zhang et al., 2022) is based on T5.

2.2.2 Pre-trained Models on Automated Program Repair

Many recent works apply pre-trained models trained on massive code corpora on APR. The dataset named bug-fix pairs (BFPs) proposed by Tufano et al. (Tufano et al., 2019) has been utilized as a benchmark for many following works. Each case in this dataset is an abstract code template processed from the raw source code dataset. (Chakraborty & Ray, 2021) summarize the raw source code of BFPs and release the raw BFPs dataset. We call the first abstract BFPs and the latter concrete BFPs. Experiment results in (Lu et al., 2021) show that CodeBERT achieves a good fix result in abstract BFPs, and the following GraphCodeBERT (Guo et al., 2020) achieves a better fix result than CodeBERT because of the additional consideration of code structure information in the pre-training stage. PLBART (Ahmad et al., 2021) and MODIT (Chakraborty & Ray, 2021) achieve better performance on BFPs. The main reason is that the model's sequence understanding and generation capacities are trained jointly because of the encoder-decoder architecture. However, the slight difference is that MODIT employs the concrete BFPs dataset instead of the abstract BFPs dataset. The latest pre-trained models (e.g., CodeT5 (Y. Wang et al., 2021), SPT-Code (Niu et al., 2022), CoditT5 (J. Zhang et al., 2022) and NatGen (Chakraborty et al., 2022), etc.) show better results on code repair and all of these latest works leverage abstract BFPs as their benchmark dataset.

Unlike previous works, we fine-tune all pre-trained models mentioned above on both abstract BFPs and concrete BFPs considering a thorough and fair comparison.

Note that most G&V-based APR approaches utilize unified benchmark datasets such as Defects4J (Just, Jalali, & Ernst, 2014) and QuixBugs (Lin, Koppel, Chen, & Solar-Lezama, 2017) for evaluation, while the DL-based APR models in this paper are evaluated on concrete BFPs and abstract BFPs. As we are not aiming to propose a new APR tool or compare the repair performance between G&V-based APR approaches and DL-based APR approaches such as CoCoNut (Lutellier et al., 2020) or CURE (N. Jiang et al., 2021), instead we focus on evaluating the performance of DL-based APR models.

2.3 Robustness Testing for Code Models

Recent efforts in NLP have shown that deep neural networks (DNNs) are vulnerable to input with small perturbations (W. E. Zhang, Sheng, Alhazmi, & Li, 2020). In the domain of deep neural networks for source code, there is a substantial line in the robustness testing of neural code models. Ramakrishnan et al. (Ramakrishnan et al., 2020) investigate the robustness of source-code models by applying k-transformation on inputs. They utilize a Bi-LSTM seq2seq model and a code2seq model as subject models and evaluate them on code summarization. They find that adversarial training with a small k can improve robustness against a stronger adversary, such as when k =1 and k=5. Rabin et al. Bielik et al. (Bielik & Vechev, 2020) verify that, similar to other domains, neural code models are vulnerable to adversarial attacks, and they find that only some parts of the input program are relevant to the model's prediction. So they abstract the rest of the programs, which makes the adversarial training more effective. Wei et al. (M. Wei et al., 2021) present a coverage-based fuzzing framework to test the robustness and generalizability of code models, where they utilize NeuralCodeSum, CODE2SEQ, and CODE2VEC as their subject models. Also, they use a random string with a fixed length to replace a variable. (Rabin et al., 2021) test the generalizability of code2vec, code2seq, and GGNN models on predicting method names. They find that these code models often fail to generalize their performance even with small semantically preserving program changes. They further find that models based on data and control dependencies in programs generalize better than neural program models based only on abstract syntax trees (ASTs). In their experiment, they design a fixed format for renaming a variable, e.g., using *var1* to replace the first

variable and using *var2* and *varn* to replace the following corresponding variables given a code snippet. Zhang et al. (H. Zhang et al., 2020) propose Metropolis-Hastings Modifier (MHM), a Metropolis-Hastings sampling-based identifier renaming approach. Instead of using a random string or fixed format for renaming a variable, as in the previous works above, MHM first pre-defines a large collection of variable names and chooses randomly from it to replace a variable. They use this identifier renaming technique to generate adversarial examples for code models. Pour et al. (Pour et al., 2021) present a search-based framework for testing the performance of code embedding models: Code2vec, Code2seq, and CodeBERT on downstream tasks. They use a synonym from the python library *nltk.corpus.wordnet* to replace the target word. However, all of the above strategies did not consider the natural context of an identifier. Following MHM (H. Zhang et al., 2020), Yang et al. (Z. Yang et al., 2022) focus on producing more natural variable substitutions for adversarial example generation. In our experiments, considering transformed code should be natural to humans and deep learning models, we apply the same strategy as (Z. Yang et al., 2022) for generating natural code transformations when renaming an identifier. We will detail how we adopt their renaming strategy in section 3.2.1.

Unlike the related work mentioned above, we are the first to focus on evaluating the performance and robustness of pre-trained models on APR. Additionally, most previous works only utilize two to three subject models. However, we consider those subject models with different architectures or pre-training objectives may have different results on robustness, so we perform a large-scale evaluation on three types of representative models: encoder-based models (e.g., CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020)), decoder-based models (e.g., CodeGPT (Lu et al., 2021)) and encoder-decoder-based models (e.g., CodeT5 (Y. Wang et al., 2021), PLBART (Ahmad et al., 2021) and SPT-Code (Niu et al., 2022)). Lastly, compared with previous works, we think that code transformation should be semantic-preserving and natural to humans and deep learning models. For example, substitutive identifiers for transformed code should not be random strings or synonyms judged only by humans. It should consider the context information before and after the substitutive identifier in the code snippet.

Chapter 3

Experiment Setup

This chapter details two experiments we designed to investigate the corresponding two research questions mentioned before. Figure 3.1 shows the experimental process. The first experiment (named "Exeriment1" in Figure 3.1) is designed to fairly evaluate the repair performance of DL-based APR models. So we fine-tune subject models for program repair based on two unified datasets: abstract BFPs and concrete BFPs. The details of these two datasets are presented in the previous section 2.2.2. Then, based on the fine-tuned models on the concrete BFPs dataset in the first experiment, the second experiment (named "Exeriment2" in Figure 3.1) further quantifies and evaluates the fine-tuned model's performance change (i.e., robustness) against different semantic-preserving transformations. Given the test dataset of concrete BFPs, we first check whether it can be parsed to an AST for each code item. Then, we check whether each code snippet is applicable for a code transformation (e.g., a code snippet without any local variables is not applicable to be changed the name of local variables). Finally, the code snippets applicable to the current code transformation are extracted for each type of code transformation to construct the original dataset. The transformed code snippets will construct the second dataset, named the transformed dataset. Section 3.2.2 will detail these two datasets' construction.

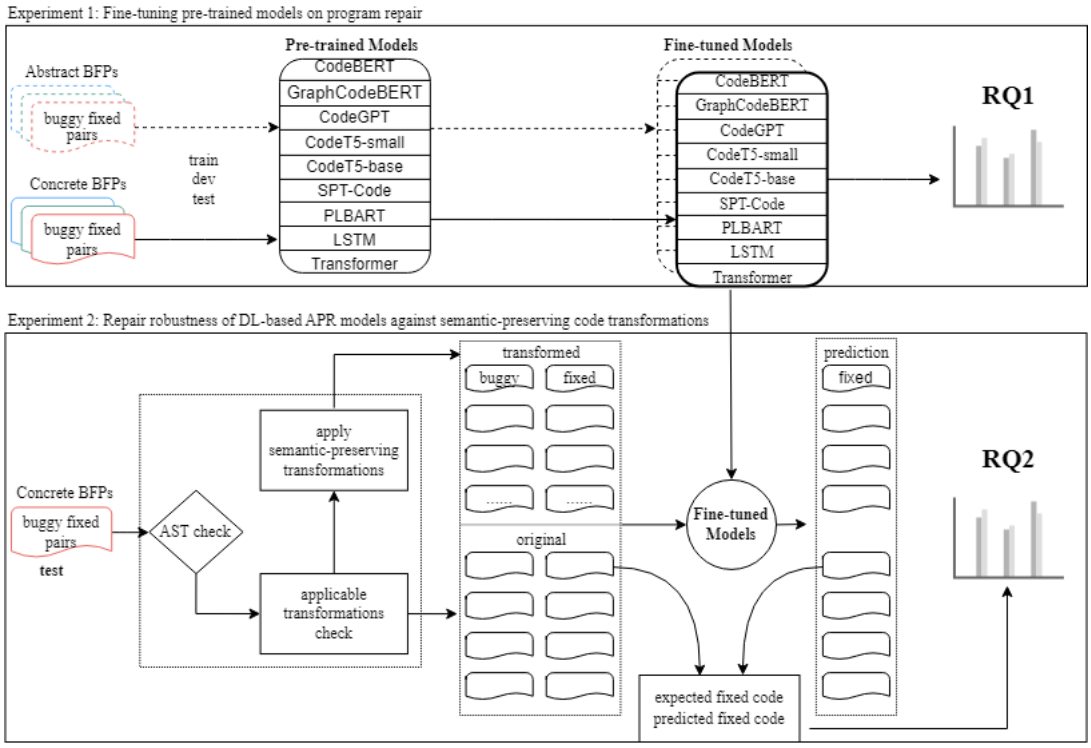

Figure 3.1: An overview of the experimental process.

3.1 Fine-tuning Pre-trained Models on Program Repair

This section presents the first experiment. We detail the subject models, hyper-parameter settings, fine-tuning datasets, and metrics we used.

3.1.1 Subject Models

We choose seven current pre-trained models as our subject models. From the perspective of three different architectures, we choose two BERT-style encoder-based models: CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020); a GPT-style decoder-based model CodeGPT (Lu et al., 2021); two BART-style encoder-decoder-based models: PLBART (Ahmad et al., 2021) and SPT-Code (Niu et al., 2022); and two versions of T5-style encoder-decoder-based models CodeT5 (Y. Wang et al., 2021), including CodeT5-small and CodeT5-base.

CodeBERT (Feng et al., 2020) is a bi-modal pre-trained model for programming languages (PL) and natural languages (NL). Following BERT (Devlin et al., 2018) and RoBERTa (Y. Liu et al., 2019), CodeBERT is developed with the multi-layer Transformer and trained on CodeSearchNet dataset (Husain, Wu, Gazit, Allamanis, & Brockschmidt, 2019). The training dataset contains 2.4M functions for six programming languages (Python, Java, JavaScript, Php, Ruby, and Go) along with natural language document pairs. CodeBERT is trained by two training objectives: masked language modeling (MLM) and replaced token detection (RTD). The first objective aims to predict the original tokens masked out given an NL-PL pair, and the second objective aims to determine which tokens in a given NL-PL pair are replaced. Experiment results introduced in CodeXGLUE (Lu et al., 2021) have shown that CodeBERT achieves good performances on various downstream tasks such as code search and clone detection. In our experiment, we fine-tune the pre-trained CodeBERT for program repair on the BFPs dataset.

GraphCodeBERT (Guo et al., 2020) has the same architecture and pre-training dataset as CodeBERT. The difference is that GraphCodeBERT also considers the inherent structure of code and utilizes the data flow in the pre-training stage. Besides using the task of masked language modeling (MLM), it introduces two structure-aware pre-training tasks: edge prediction and node alignment. The first is to predict code structure edges, and the other is to align representations

between source code and code structure. Experiment results introduced in CodeXGLUE (Lu et al., 2021) have shown that GraphCodeBERT performs better than CodeBERT on four downstream tasks, i.e., code search, clone detection, code translation, and code refinement. In our experiment, we first fine-tune it on the BFPs dataset, and then we use the fine-tuned GraphCodeBERT to infer the fix prediction given the buggy code.

CodeGPT (Lu et al., 2021) is a pre-trained programming language model for sequence-to-sequence code generation tasks. It has the same model architecture (i.e., a single left-to-right decoder) and training objectives of GPT-2 (Radford et al., 2019), which consists of twelve layers of Transformer decoders. CodeGPT is trained on Python and Java corpora from the CodeSearchNet dataset (Husain et al., 2019), which includes 1.1M Python functions and 1.6M Java methods. Recent work done by Jiang et al. (N. Jiang et al., 2021) showed the effectiveness of GPT for the program repair task. In our experiment of program repair, we fine-tune the pre-trained CodeGPT for program repair on the BFPs dataset.

CodeT5 (Y. Wang et al., 2021), based on T5 (Raffel et al., 2020), is an encoder-decoder pre-trained model. Its pre-training datasets contain CodeSearchNet and C/CSharp datasets from BigQuery (*BigQuery*, n.d.), totaling 8.35 million instances. For the pre-training tasks, besides Masked Span Prediction (MSP), which is similar to T5, CodeT5 adds two additional code-specific tasks, i.e., Identifier Tagging (IT) and Maksed Identifier Prediction (MIP), to complement the denoising sequence-to-sequence pre-training. CodeT5 has shown promising performance on code-related understanding and generation tasks such as code summarization, code generation, and clone detection. Our experiment will use the CodeT5-small and CodeT5-base as subject models for program repair.

PLBART (Ahmad et al., 2021), is an encoder-decoder pre-trained model based on BART (Lewis et al., 2019). The pre-training dataset contains 470M Java and 210M Python functions collected from GitHub and 47M posts in natural language descriptions (English) collected from Stack Overflow. The pre-training objective is to reconstruct the original input sequence given an input sequence that is corrupted by a noise function. Their experimental results show that PLBART performs better than both CodeBERT and GraphCodeBERT on four downstream tasks, including code summarization, code generation, and code translation. In our experiments, we utilize their pre-trained PLBART and fine-tune it for program repair on the BFPs dataset.

SPT-Code (Niu et al., 2022) is also an encoder-decoder pre-trained model based on BART. It is pre-trained on the CodeSearchNet dataset. The pre-training tasks include the Masked Sequence-to-Sequence prediction, new-designed Code-AST Prediction (CAP), and Method Name Generation (MNG). SPT-Code's input contains three components: code tokens, linearized AST, and natural language extracted only from code (i.e., method name and API call sequence). SPT-Code achieves promising performances on five downstream tasks, including code summarization, code completion, code translation, code search, and code repair. The results show that SPT-Code achieves the SOTA performance compared with CodeBERT and GraphCodeBERT on the five tasks mentioned above. We employ the pre-trained SPT-Code in our experiments and fine-tune it for program repair on the BFPs dataset. We further compare its repair performance and robustness with other subject models, which is never conducted in previous works.

Besides the pre-trained models above, we also configure two models trained from scratch for comparison: **LSTM-based NMT model** (Bahdanau, Cho, & Bengio, 2014), and **Transformer-based NMT model** (Vaswani et al., 2017). The former LSTM-based NMT model is configured with a relatively simple architecture. It contains one encoder layer and one decoder layer with an attention module. The latter Transformer-based NMT model is configured with similar architecture as most of the pre-trained models above, which are twelve encoder layers and twelve decoder layers. Both of them are trained from scratch based on our datasets. Table 3.1 and 3.2 illustrate the training settings for all subject models.

Table 3.1: Statistics of subject models (1).

Group	Models	Architecture	Number of Parameters	Number of Layers	Attention Heads
BERT-style	CodeBERT (2020)	Encoder	125 M	12	12
	GraphCodeBERT (2020)	Encoder	125 M	12	12
GPT-style	CodeGPT (2021)	Decoder	124 M	12	12
T5-style	CodeT5-small (2021)	Encoder-Decoder	60 M	6	8
	CodeT5-base (2021)	Encoder-Decoder	220 M	12	12
BART-style	PLBART (2021)	Encoder-Decoder	140 M	6	12
	SPT-Code (2022)	Encoder-Decoder	262 M	12	12
Training from Scratch	LSTM-based	Encoder-Decoder	82 M	1	1
	Transformer-based	Encoder-Decoder	406 M	12	12

Table 3.2: Statistics of subject models (2). Following pre-training tasks are listed in the table: Masked Language Modeling (MLM), Replaced Token Detection (RTD), Edge Prediction (EP), Node Alignment (NA), Causal Language Modeling (CLM), Masked Span Prediction (MSP), Identifier Tagging (IT), Masked Identifier Prediction (MIP), Code-AST Prediction (CAP), Masked Sequence to Sequence (MASS), Method Name Generation (MNG), Denoising Autoencoding (DAE).

Group	**Models**	**Pre-training Dataset**	**Pre-training Tasks**
BERT-style	CodeBERT (2020)	CodeSearchNet	MLM, RTD
	GraphCodeBERT (2020)	CodeSearchNet	MLM, EP, NA
GPT-style	CodeGPT (2021)	CodeSearchNet	CLM
T5-style	CodeT5-small (2021)	CodeSearchNet and C/CSharp from BigQuery	MSP, IT, MIP
	CodeT5-base (2021)	CodeSearchNet and C/CSharp from BigQuery	MSP, IT, MIP
BART-style	PLBART (2021)	Java/Python from GitHub and Posts from Stack Overflow	DAE
	SPT-Code (2022)	CodeSearchNet	CAP, MASS, MNG

Table 3.3: Fine-tuning configuration of subject models (1).

Group	**Models**	**Optimizer**	**Tokenizer**	**Maximum Epoch**	**Maximum Length**
BERT-style	CodeBERT (2020)	AdamW	BPE	30	256
	GraphCodeBERT (2020)	AdamW	BPE	30	256
GPT-style	CodeGPT (2021)	AdamW	BPE	30	512
T5-style	CodeT5-small (2021)	AdamW	BPE	50	130
	CodeT5-base (2021)	AdamW	BPE	50	130
BART-style	PLBART (2021)	Adam	SentencePiece	30	512
	SPT-Code (2022)	AdamW	BPE	50	256
Training from Scratch	LSTM-based NMT	Adam	SentencePiece	30	500
	Transformer-based NMT	Adam	SentencePiece	30	500

Table 3.4: Fine-tuning configuration of subject models (2).

Group	Models	Learning Rate	Batch Size (Training)	Beam Size
BERT-style	CodeBERT (2020)	5e-5	16	5
	GraphCodeBERT (2020)	5e-5	16	5
GPT-style	CodeGPT (2021)	5e-5	16	5
T5-style	CodeT5-small (2021)	5e-5	32	10
	CodeT5-base (2021)	5e-5	32	10
BART-style	PLBART (2021)	5e-5	16	5
	SPT-Code (2022)	5e-5	64	5
Training from Scratch	LSTM-based NMT	1e-3	32	5
	Transformer-based NMT	5e-5	32	5

3.1.2 Hyper-parameter Settings in Fine-tuning

Considering the best hyper-parameter settings for each model, for CodeBERT, GraphCodeBERT, CodeGPT, and PLBART, we employ the same hyper-parameter settings in (Chakraborty & Ray, 2021). Specifically, we use Label Smoothed Cross Entropy (Müller, Kornblith, & Hinton, 2019) as the loss function. We set the learning rate to 5e-5 with Adam optimizer. We set the number of maximum training epochs as 30. For CodeT5 and SPT-Code, we employ the hyper-parameter settings of original works (Niu et al., 2022; Y. Wang et al., 2021) respectively. Table 3.3 and 3.4 list the detailed hyper-parameter configuration of all subject models. Note that we try to keep the same hyper-parameter settings with the corresponding original works unless we meet limitations, as such settings have experimented well. Finally, we consider such settings suitable for fine-tuning models (e.g., GPU memory limitation discussed in section 5.1).

We employ 1×Nvidia Tesla V100 GPU with 32GB memory for all fine-tuning infrastructure.

3.1.3 Datasets

There exist two different versions of the datasets mainly used in most of the recent DL-based APR models (Ahmad et al., 2021; Chakraborty et al., 2022; Chakraborty & Ray, 2021; Niu et al., 2022; Y. Wang et al., 2021; J. Zhang et al., 2022), called BFPs (bug-fix pairs) initially proposed by (Tufano et al., 2019) in their program repair work. Each pair in BFPs is composed of a tuple *(mb, mf)*, where *mb* presents a buggy code component, *mf* presents the corresponding fixed code. The two existing versions of BFPs are abstract BFPs and concrete BFPs. Table 3.5 provides an

example of these two different datasets versions. Both datasets contain method-level pairs of a buggy and corresponding fixed code extracted from bug-fixing commits in thousands of GitHub Java repositories. The abstract BFPs are used originally in (Tufano et al., 2019)'s experiment, and all the code is abstracted from the raw source code to the abstract template code. The concrete BFPs are initially from Tufano's work and further slightly processed by (Chakraborty & Ray, 2021). Both datasets contain two sub-datasets: B2Fs-small and B2Fs-medium; the difference is that in the former dataset, the maximum token length of methods is 50, and in the latter dataset, the methods are no longer than 100 tokens in length. Specifically, MODIT employs concrete BFPs for evaluation, while PLBART, SPT-Code, and Code-T5 employ abstract BFPs for evaluation.

Table 3.5: An example of abstract BFPs and concrete BFPs. The code snippet of concrete BFPs is raw source code, and that of abstract BFPs is a code template extracted by (Tufano et al., 2019).

abstract BFPs

```
public void METHOD_1(){
    VAR_1.METHOD_2(VAR_2);
    VAR_2 = null;
}
```

concrete BFPs

```
public void unregisterNSDService(){
    mNsdManager.unregisterService(
        networkRegistrationListener
        );
    networkRegistrationListener =
        null;
}
```

As our motivation here is to evaluate the repair performance of different models, considering a fair experiment on the comparison, we choose abstract BFPs and concrete BFPs as the unified repair dataset to fine-tune all the pre-trained models. For abstract BFPs, we use the original split as used in (Tufano et al., 2019), and for concrete BFPs, we reuse the same split as used in (Chakraborty & Ray, 2021). Both abstract BFPs and concrete BFPs are split into three parts: training, validation, and testing datasets. Training and validation datasets are used in fine-tuning, and the testing dataset is used for evaluation. Note that in the following experiment 3.2, we will only focus on concrete BFPs; the consideration behind this is that we believe deep learning-based repair approaches should be end-to-end and try to avoid adding additional manual efforts as much as possible (i.e., transforming

raw source code to abstract version for model's input, then transforming the output abstract code to raw source code). Table 3.6 provides the statistics of fine-tuning datasets.

Table 3.6: Statistics of fine-tuning datasets. Train#, Valid#, and Test# stand for the number of the training, validation, and testing datasets. As concrete BPFs are slightly processed by (Chakraborty & Ray, 2021) based on abstract BFPs, the size of concrete BPFs is slightly less than abstract BFPs.

Group	Dataset	Train#	Valid#	Test#
Concrete BPFs	BFPs-small	46628	5828	5831
	BFPs-medium	53324	6542	6538
Abstract BPFs	BFPs-small	46680	5835	5835
	BFPs-medium	52364	6546	6545

3.1.4 Metrics

We choose Accuracy@1 and CodeBLEU to evaluate the correctness of DL-based APR predictions.

Accuracy@1

Following previous works (Ahmad et al., 2021; Chakraborty et al., 2022; Chakraborty & Ray, 2021; Guo et al., 2020; Lu et al., 2021; Niu et al., 2022; Y. Wang et al., 2021; J. Zhang et al., 2022), we also use Accuracy@1 (Acc@1) as the primary metric for our experiments. Specifically, in the model's inference stage, if the generated candidate (i.e., the predicted fixed code) with the highest probability (i.e., the model will generate a descending ranked list of candidates code based on probability, we choose the top one) is identical to the goal sentence (i.e., the expected fixed code), we consider it correct, otherwise incorrect.

CodeBLEU

BLEU (Papineni, Roukos, Ward, & Zhu, 2002) (bilingual evaluation understudy) is an algorithm for evaluating the quality of a machine-translated text from the text, which is used in the natural language processing field. CodeBLEU (Ren et al., 2020) is a popular metric for code processing-related

tasks as it can consider not only the text similarity (i.e., n-gram match in NLP) but also the similarity of code syntax and data flow. This value indicates how similar the prediction code is to the ground truth, the more values closer to one, the more similar they are. The following formula calculates CodeBLEU: *CodeBLEU = alpha * ngram_match_score + beta * weighted_ngram_match_score + gamma * syntax_match_score + theta * dataflow_match_score*, where alpha, beta, gamma, and theta are all 0.25. We reuse the implementation of CodeBLEU calculation in (Lu et al., 2021).

3.2 Repair Robustness of DL-based APR Models against Semantic-preserving Code Transformations

This section presents the second experiment. We detail the design of semantic-preserving code transformations and how we construct our datasets for repair robustness evaluation on DL-based APR models.

3.2.1 Semantic-preserving Code Transformations

Previous works study code refactoring in practice (Golubev, Kurbatova, AlOmar, Bryksin, & Mkaouer, 2021; Negara, Chen, Vakilian, Johnson, & Dig, 2013; Negara, Vakilian, Chen, Johnson, & Dig, 2012; Tsantalis, Ketkar, & Dig, 2020). However, the semantic-preserving code transformations in this thesis are not code refactorings. Recent studies (Pour et al., 2021; M. Wei et al., 2021; Z. Yang et al., 2022) utilize code transformations for robustness testing or adversarial attack of code models, while code refactorings are mainly employed for optimizing code in the practice of software development. We leave how code refactoring impacts the performance of DL-based APR models as future work.

We totally choose nine types of different semantic-preserving code transformations.

- **Local variable renaming** is a semantic-preserving transformation that renames all the occurrences of the first local variable. The new name of the variable is generated by the pre-trained model. The details will be presented in the following paragraphs.
- **Method renaming** is a semantic-preserving transformation that renames the method. The new name is generated by the same strategy as local variable renaming.
- **Parameter renaming** is a semantic-preserving transformation that renames the parameter. The new name is generated by the same strategy as local variable renaming.
- **Insert log statement** is a semantic-preserving transformation that adds *System.out.println("log");* as the first code statement in a method-level code snippet.

- **Insert try catch** is a semantic-preserving transformation that adds a *try-catch* in code statements.
- **Boolean exchange** is a semantic-preserving transformation that swaps *true* with *false* and vice versa.
- **Loop exchange** is a semantic-preserving transformation that replace *for* with *while* and vice versa.
- **Convert switch to if** is a semantic-preserving transformation that replace *if-else* statements with *switch-case* statements and vice versa.
- **Reorder condition** is a semantic-preserving transformation that swaps the conditions before and after == and *!*=.

There are several alternative strategies for the renaming transformation, and we describe these in detail in section 2.3. In this section, we will detail why we adopt (Z. Yang et al., 2022) for our renaming transformations and how we implement it in our experiment. As mentioned in the related work chapter 2.3, M. Wei et al. (2021) use a random string with a fixed length to replace a variable; Rabin et al. (2021) design a fixed format for renaming a variable; H. Zhang et al. (2020) propose Metropolis-Hastings Modifier (MHM) to rename an identifier; Pour et al. (2021) use a synonym from the python library *nltk.corpus.wordnet* to replace the target word. However, all of the above strategies did not consider the natural context of an identifier. Renaming by random string or fixed format does not comply with common code conventions in the real world, and developers may reject the transformed unnatural code. When it comes to pre-trained models, their knowledge is based on pre-training on large human code repositories, which means they also follow the common code conventions in the real world. Additionally, suppose we feed a buggy code with extremely long string variables or common variable names to the model, and its tokenizer may tokenize them to completely different sub-tokens, which will impact the model's prediction. Finally, we adopt the recent naturalness-aware substitution algorithm proposed by (Z. Yang et al., 2022). They aim to generate a variable substitution natural to developers and models, to support their adversarial attack in their experiment. We thank this work for its open-source experiment code.

Algorithm 1 introduces how we employ the naturalness-aware substitution algorithm (Z. Yang et al., 2022) for our renaming strategy. First, we extract the first variable (*var*) by an input source code (Line 1). Then, given a code snippet with the masked token (*var*), the masked language prediction function (*mask_lang_pred()*) can predict a list of substitutions (*subs*) based on the input context (Line 3). Then, it leverages the contextualized embedding function (*embedding()*) to compute the embedding of the original masked token (*embedding_var*) and corresponding substitutive tokens (*embedding_sub*) in the substitution list (Line 4-7). Next, it computes the Cosine similarity (cosine_var_subs) of the embedding of the original token with each substitution in its substitutions list (Line 8-9). Then, it sorts all the substitutions in descending order through the Cosine similarity value (Line 11). Finally, we select the top one substitution to replace the variable in all occurrences of a code snippet and return the transformed code snippet (*c'*) (Line 11-14).

Algorithm 1: Naturalness Aware Substitution for Renaming-related Transformations

Input: c: input source code, M: pre-trained model

Output: c': transformed code

1 $var \leftarrow extract(c)$;

2 # extract(c) returns the first variable (same as the parameter, method) in code c

3 $subs \leftarrow mask\_lang\_pred(var, c, M)$;

4 $embedding\_var \leftarrow embedding(var, M)$;

5 $cosine\_var\_and\_subs \leftarrow \emptyset$;

6 **for** *sub in subs* **do**

7 $\quad embedding\_sub \leftarrow embedding(sub, M)$;

8 $\quad cosine\_var\_sub \leftarrow cosine\_similarity(embedding\_var, embedding\_sub)$;

9 $\quad cosine\_var\_subs.add(cosine\_var\_sub)$;

10 **end**

11 $var\_sub\_top \leftarrow sort(cosine\_var\_subs).top\_one$;

12 $c' \leftarrow replace(var\_sub\_top, var.occurrences, c)$;

13 # var.occurrences returns all occurrences of var

14 **return** c';

This algorithm leverages two functions of CodeBERT or GraphCodeBERT: masked language

prediction and contextualized embedding. Given a code snippet with masked tokens, the masked language prediction function can predict a list of potential substitutions based on the input context. Then, it leverages the contextualized embedding function to compute the embedding of the original masked tokens and corresponding substitutive tokens in the substitution list. Next, it computes the Cosine similarity of the embedding of the original token with each substitution in its substitutions list. Finally, it sorts all the substitutions in descending order through the Cosine similarity value.

We utilize tree-sitter (*tree-sitter*, n.d.) to parse java code and identify the target position for renaming an identifier. We utilize this algorithm only for generating renaming substitutions for an identifier. Note that we apply this algorithm not only on local variable renaming but also on parameter and method renaming. Considering the complexity, we only rename one variable and one parameter if there are multiple ones in a code snippet. As different code snippets have different numbers of variables, the number of renamed variables for different code snippets may introduce bias for our experiment. We leave it as future work about investigating how the different number of renamed identifiers impact the repair effect of DL-based APR models. We only choose the top one substitution of a variable, parameter, or method from the ranked substitution list, as this substitute is what pre-trained models think can fit the code context best. Different from the previous work (Z. Yang et al., 2022), which generates top-k (k=60 in their experiment) ranked substitutions for each variable. They iterate these substitutions to attack the model until one adversarial sample is generated.

List 1 gives an example of substitutes generated by GraphCodeBERT for a code snippet. Given a code snippet, we first locate that the local variable's name is *temp*, the method's name is *incrementKey*, and the parameter's name is *key*. Next, we provide the top five substitutes for each name of a local variable, method, and parameter. We can find that the top one substitute of *incrementKey* is *"incrementalIndex"*, which are semantically close to each other. However, the top one substitute of *key* is *"is"*, and that of *temp* is *"count"*, both are not semantically close to the original tokens. It verifies that the pre-trained models generate the masked tokens by all the context information instead of only one original token in a code snippet. We can find the words in both pairs of (*key*, *is*) and (*temp*, *count*) are not semantically similar, as the substitutions are predicted by the variable-surrounded context instead of the variable itself.

Table 3.7 gives examples of the original code snippet and the corresponding renamed code snippet. For the local variable renaming, method renaming, and parameter renaming, we apply the top-1 substitution as shown in List 1 to replace the corresponding variable and parameter occurrences. For example, for renaming a local variable, after we replace the *temp* by *"count"* in the code statement *int count = keys.get(key);*, the following code statements which using *temp* will also be changed, i.e., changing *keys.set(key, (temp++));* to *keys.set(key, (count++));*. For renaming a parameter, the logic is the same as renaming a local variable. While for renaming a method, we only need to change once as all items in our dataset are a single method-level code snippet, and each code snippet only contains one method.

Note that we only rename the first variable or parameter and all of their occurrences in a method-level code for renaming a variable or parameter. We leave transformations about renaming multiple variables and parameters as future work.

For the other six types of semantic-preserving code transformations, we thank the work (Rabin, Wang, & Alipour, 2019) that open source their code, and we adopt their tool to generate our six types of semantic-preserving code transformations. However, as their tool only accepts Java files as input, we first convert each method-level code snippet in the testing dataset to one Java file. So the number of Java files is the same as the size of the converted dataset. Then, we directly use their tool to generate six types of semantic-preserving transformation files for each Java file. Finally, we gather transformed Java files to the dataset file, i.e., each Java file is used as one line of code to construct transformation datasets. Note that (Rabin et al., 2019) is based on Javaparser to build AST for code, while Javaparser is not suitable for parsing a partial code snippet, so the tool adds a *Class T* {} head to each method-level code snippet before applying transformations, and convert it to its original code snippet (i.e., delete *Class T* {} head) after transformations.

Table 3.7 and 3.8 provide examples of these six code transformations. For the examples of insert log statement and insert try catch, we add a statement *System.out.println("log");* in the first line and add *try-catch* in the code statement; For the example of boolean exchange, we change *boolean res = false;* to *boolean res = true;*, also change *return res;* to *return !(res);* to ensure the semantic equivalence; For the example of loop exchange, we use *while* to replace *for* and add *int i = 1;* in line 2 to keep the semantic equivalence; For the example of convert switch to if, we use *if-else* to replace

```
{
    "code":
        "public void incrementKey(int key) {
            int temp = keys.get(key);
            keys.set(key, (temp++));
        }",
    "substitutes":{
        "incrementKey":[
            "incrementalIndex",
            "increaseKey",
            "IncreaseInt",
            "IncreaseKey",
            "IncreaseIndex",
            ......
        ],
        "key":[
            "is",
            "primary",
            "ey",
            "table",
            "y",
            ......
        ],
        "temp":[
            "count",
            "max",
            "pre",
            "partial",
            "random",
            ......
        ]
    }
}
```

Listing 1: An example of substitutes generated by GraphCodeBERT for local variables, methods, and parameters.

switch-case; For the example of reorder condition, we swap the order of conditions, i.e., changing *(this.parent)==null* to *null==(this.parent)*.

Table 3.7: Examples of semantic-preserving code transformations of local variable renaming, method renaming, parameter renaming, insert log statement and insert try catch.

original code

```
1 public void incrementKey(int
      key) {
2     int temp = keys.get(key);
3     keys.set(key, (temp++));
4 }
```

local variable renaming

```
1 public void incrementKey(int key
      ) {
2     int count = keys.get(key);
3     keys.set(key, (count++));
4 }
```

method renaming

```
1 public void incrementalIndex(int
      key) {
2     int temp = keys.get(key);
3     keys.set(key, (temp++));
4 }
```

parameter renaming

```
1 public void incrementKey(int
      is) {
2     int temp = iss.get(is);
3     iss.set(is, (temp++));
4 }
```

insert log statement

```
1 public void incrementKey(int key
      ) {
2     System.out.println("log");
3     int temp = keys.get(key);
4     keys.set(key, (temp++));
5 }
```

insert try catch

```
1 public void incrementKey(int key
      ) {
2     int temp = keys.get(key);
3     try {
4         keys.set(key, (temp++));
5     } catch (Exception ex) {
6         ex.printStackTrace();
7     }
8 }
```

Table 3.8: Examples of semantic-preserving code transformation of boolean exchange, loop exchange, convert switch to if and reorder condition.

original code

```
1 public boolean contains(int ID) {
2     E e = getElementByID(ID);
3     boolean res = false;
4     if (e != null) {
5         res = true;
6     }
7     return res;
8 }
```

boolean exchange

```
1 public boolean contains(int ID) {
2     E e = getElementByID(ID);
3     boolean res = true;
4     if (e != null) {
5         res = false;
6     }
7     return !(res);
8 }
```

original code

```
1 private void fillTower(int N) {
2     for (int i = 1; i <= N; i++)
          {
3         towers.get(0).push(i);
4     }
5 }
```

loop exchange

```
1 private void fillTower(int N) {
2     int i = 1;
3     while (i <= N) {
4         towers.get(0).push(i);
5         i++;
6     }
7 }
```

original code

```
1 public void onClick(View view) {
2     switch (view.getId()) {
3         case R.id.signInButton:
4             signIn(); break;
5         case R.id.signOutButton:
6             signOut(); break;
7     }
8 }
```

convert switch to if

```
1 public void onClick(View view) {
2     if (view.getId() == R.id.
          signInButton) {
3         signIn();
4     } else if (view.getId() == R
          .id.signOutButton) {
5         signOut();
6     }
7 }
```

original code

```
1 public boolean hasParent() {
2     return (this.parent)==null ?
          false : true;
3 }
```

reorder condition

```
1 public boolean hasParent() {
2     return null==(this.parent) ?
          false : true;
3 }
```

3.2.2 Dataset Construction

In this experiment, we only use concrete BFPs for performing code transformations, as code snippets in abstract BFPs are abstract code templates instead of raw source code, which is unsuitable for transformations. Table 3.5 gives an example of abstract code and the corresponding concrete code. We can find all variables, methods, and parameters are converted to the fixed format such as *VAR_1*, *METHOD_1*, and *PARAMETER_1*. Such transformations are not semantically natural to developers in the real world. We only choose concrete BFPs for performing semantic-preserving code transformations as the code snippets in this dataset are raw and natural source code. Also, the transformed code snippets for concrete BFPs are more natural than that for abstract BFPs, which is more meaningful for the robustness evaluation of models.

We construct two datasets for each type of semantic-preserving code transformation: the original dataset and the transformed dataset. Specifically, the original dataset of each transformation is extracted from the test dataset of concrete BFPs before applying the code transformation because not each code transformation is applicable for each code snippet. Note that we only use the testing dataset from concrete BFPs for this experiment. The training and validation datasets are used in the fine-tuning stage. For instance, for renaming a local variable in the iteration stage of all items in the test dataset of concrete BFPs, if the current code snippet does not contain any local variable, which means this code snippet is not suitable for being applied the variable renaming transformation, so we skip it. Otherwise, we extract this code item and add it to the original dataset, then add its transformed code to the corresponding transformed dataset. Finally, we generate nine datasets groups. For both BFPs-small and BFPs-medium (section 3.1.3 explains their difference), each group contains one type of transformed dataset and the corresponding original dataset. In each group, the number of code items in the original and transformed datasets is the same. Table 3.9 lists the statistics of nine groups of constructed datasets.

Table 3.9: Statistics of transformed datasets and corresponding original datasets. Each transformed code snippet corresponds to the original code snippet, so the number of each type of the transformed dataset and the corresponding original dataset is the same.

Transformations	**BFPs-small**	**BFPs-medium**
local_variable_renaming	251	1099
method_renaming	5674	6217
parameter_renaming	3444	4302
boolean_exchange	32	146
convert_switch_to_if	44	166
insert_log_statement	5608	6384
insert_try_catch	3321	5073
loop_exchange	65	708
reorder_condition	1125	3578

Chapter 4

Experiment Results and Analysis

4.1 RQ1: What Is the Repair Performance of Different DL-based APR Models?

4.1.1 Motivation

There are mainly three types of pre-trained models from the architectural perspective: BERT-style encoder-based models, GPT-style decoder-based models, and T5-style and BART-style encoder-decoder-based models. We aim to investigate their repair performance based on a fair comparison experiment. Note that previous works have adopted pre-trained models on program repair. While some models employ abstract BFPs dataset, others use concrete BFPs dataset. Some also utilize the dataset extracted by themselves from big software repositories. However, a fair study is needed to compare the repair performance of all the representative models based on unified datasets. The motivation is that we try to reveal quantitatively which representative models perform better for APR. For example, SPT-Code and Code-T5 evaluate the model's capacity of program repair only on abstract BFPs. To the best of our knowledge, no existing works evaluate these models on concrete BFPs. The concrete BFPs contains the raw source code instead of the abstract code template in abstract BFPs. We argue that fine-tuning models on raw code datasets are better than abstract code datasets because such fine-tuned models can directly infer fixed code for a buggy code. While the

models fine-tuned on abstract code datasets need a set of manual efforts. Such manual efforts contain pre-processing from raw code to abstract code when feeding it to models and post-processing from abstract code to raw code after obtaining the models' output. Also, there needs to be more work comparing the repair performance of DL-based APR models on abstract BFPs and concrete BFPs.

Additionally, these models leverage program repair as one of the downstream tasks to evaluate the model's understanding and generation capability. However, we focus on one task-specific comparison from the perspective of program repair, i.e., which models have better repair performance. Compared with previous works, we use more representative models and evaluate them uniformly on both concrete BFPs and abstract BFPs instead of either of them.

Lastly, given one model, we wonder which type of code format (i.e., abstract or concrete source code) is more suitable for DL-based APR models.

4.1.2 Approach

We conducted our first experiment to answer this research question. Section 3.1 describes the experiment setup. We first download all the pre-trained checkpoints of all subject models. Note that at this stage, these pre-trained models have the understanding and generation capacity of natural language or programming language (i.e., source code), and their task-specific capability needs to be fine-tuned on the corresponding dataset. We respectively employ concrete BFPs and abstract BFPs for fine-tuning all subject models. Table 3.3 and 3.4 detail the fine-tuning setups for hyper-parameters of all subject models. Table 4.1 shows our detailed result for different fine-tuned models on concrete BFPs and abstract BFPs. Section 3.1.3 details the evaluation datasets, and Table 3.6 lists the statistics of split datasets for both abstract BFPs and concrete BFPs (i.e., training, validation, and testing datasets).

4.1.3 Results and Discussion

Most encoder-decoder-based APR models (SPT-Code, CodeT5, and PLBART) and the decoder-based APR model (CodeGPT) have better repair accuracy than encoder-based APR models (CodeBERT and GraphCodeBERT) on both abstract BFPs and concrete BFPs. Only

Table 4.1: Repair performance of subject models fine-tuned on concrete BFPs and abstract BFPs. "acc" stands for Accuracy@1(%) and "codebleu" stands for CodeBLEU(%).

Group	Models	Dataset	Abstract BFPs		Concrete BFPs	
			acc	**codebleu**	**acc**	**codebleu**
BERT-style	CodeBERT	BFPs-small	15.32	78.26	16.68	80.51
		BFPs-medium	3.00	91.27	8.88	87.72
	GraphCodeBERT	BFPs-small	14.82	78.77	17.62	79.64
		BFPs-medium	3.85	91.19	6.65	87.20
GPT-style	CodeGPT	BFPs-small	19.61	74.30	22.20	80.08
		BFPs-medium	12.25	84.63	11.53	86.66
T5-style	codeT5-base	BFPs-small	21.61	77.84	25.29	79.73
		BFPs-medium	13.23	89.72	15.49	87.78
	codeT5-small	BFPs-small	20.34	77.92	21.76	78.10
		BFPs-medium	11.82	89.42	12.92	86.29
BART-style	SPT-Code	BFPs-small	17.06	73.14	22.79	82.59
		BFPs-medium	11.34	88.16	20.56	89.19
	PLBART	BFPs-small	17.60	77.49	18.40	77.64
		BFPs-medium	8.71	87.61	6.13	87.15
Training from Scratch	LSTM-based	BFPs-small	6.24	74.44	2.44	49.75
		BFPs-medium	2.15	86.87	0.69	77.36
	Transformer-based	BFPs-small	12.00	73.66	5.61	62.55
		BFPs-medium	5.99	84.50	2.00	73.80

on BFPs-medium of concrete BFPs, PLBART (6.13%) is slightly lower than GraphCodeBERT (6.65%). From Table 4.1 and Figure 4.1, when we compare different models' performance on concrete BFPs, we find the best repair effect on BFPs-small is generated by the CodeT5-base model, which is 25.29%. The following SPT-Code (22.79%), CodeT5-small (21.76%), and PLBART (18.40%) also show good performance. Additionally, CodeGPT's result (22.20%) is close to SPT-Code, while other encoder-based models (CodeBERT and GraphCodeBERT) are all lower than 20%. Regarding BFPs-medium, the best repair effect is from SPT-Code (20.56%). The following models' results are significantly lower than SPT-Code, which are 12.92%, 11.53%, 8.88%, 6.65%, and 6.13% for CodeT5-small, CodeGPT, CodeBERT, GraphCodeBERT, and PLBART respectively. Overall, most encoder-decoder-based APR models and the decoder-based APR model have better repair accuracy than encoder-based APR models.

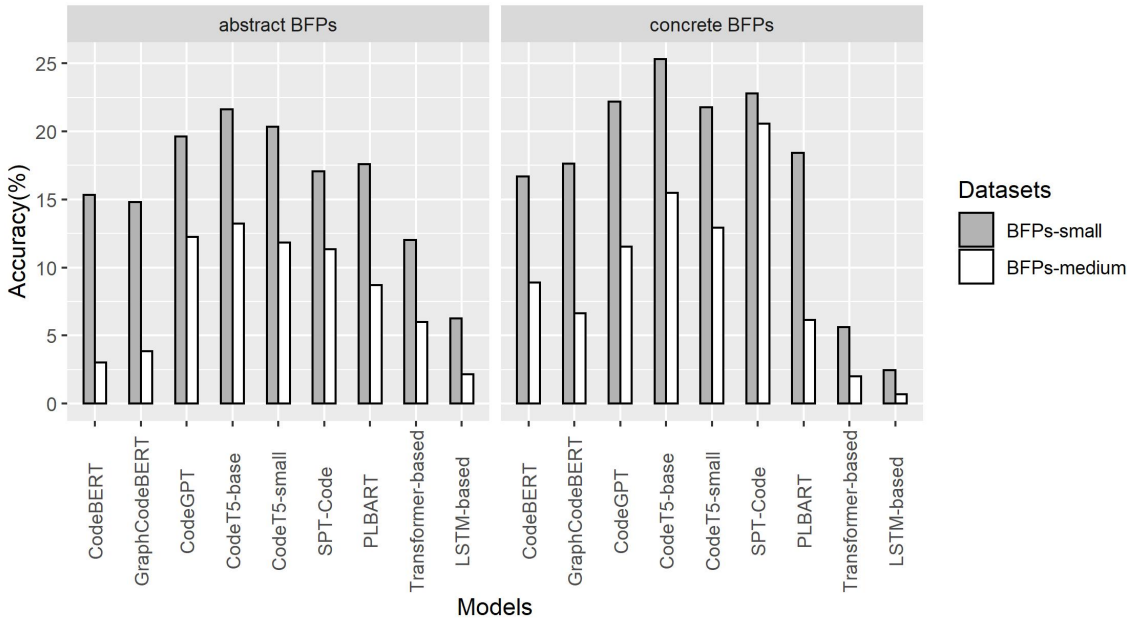

Figure 4.1: The repair accuracy of subject models on abstract BFPs and concrete BFPs.

The reason is, for the encoder-decoder-based models, the encoder and decoder components can learn jointly and build the capability of both code understanding and generation from their pre-training stages. However, in an encoder-based model such as CodeBERT, even if a decoder is attached, this decoder is actually trained from scratch in the fine-tuning stage. So it needs to

gain knowledge from a more general and larger pre-training dataset based on different pre-training objectives due to the lack of the pre-training stage. The decoder-based model is better than encoder-based models because its natural architecture is designed for sequence generation tasks (e.g., code repair and code generation) instead of sequence understanding tasks (e.g., code search).

Most DL-based APR models fine-tuned on the concrete BFPs have better repair performance than those fine-tuned on the abstract BFPs. However, in models trained from scratch, this result is reversed. Only for CodeGPT and PLBART on concrete BFPs-medium, the accuracy is both slightly lower than them on abstract BFPs-medium (Figure 4.2). Particularly, we can see from Table 4.1 and Figure 4.2, the repair accuracy of SPT-Code on concrete BFPs is 20.56%, while that on abstract BFPs is 11.34%. Similarly, CodeBERT, GraphCodeBERT, CodeT5, and PLBART show better repair results on concrete BFPs than on abstract BFPs. While CodeGPT's and PLBART's repair accuracy on concrete BFPs-medium (11.53% and 6.13%, respectively) are not better than that on abstract BFPs-medium (12.25% and 8.71%, respectively), the gap is small. However, for the LSTM-based NMT model and the Transformed-based model, their repair accuracy on abstract BFPs is better than that on concrete BFPs.

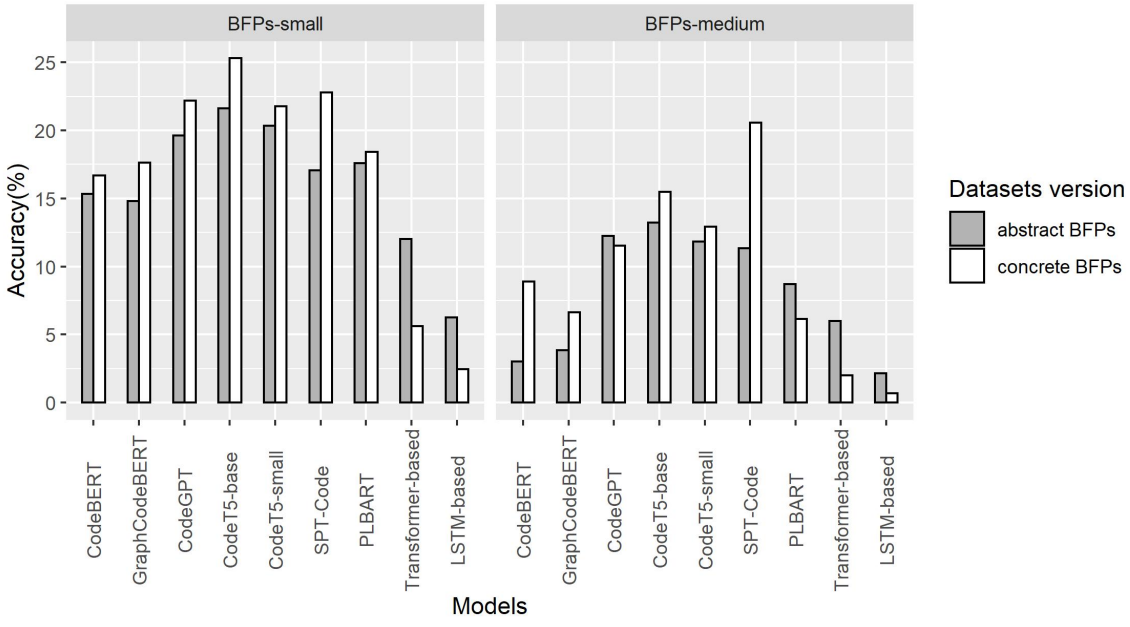

Figure 4.2: The repair accuracy comparison of subject models on abstract BFPs and concrete BFPs.

To explain these results, we can think about the pre-training datasets such as CodeSearchNet (Husain et al., 2019), all the cases in the training dataset are concrete source code instead of abstract source code, which means the models only learn the general knowledge on concrete source code in the pre-training stage. While the results on the LSTM-based NMT model and the Transformed-based model indicate that the model only trained on abstract code may have better repair results than that only trained on concrete code. What would things be like if we processed and transformed all concrete source code to an abstract version for the pre-training dataset? However, the deep learning-based approaches should be end-to-end and try to avoid adding additional manual efforts as much as possible (i.e., transforming a bug-fix pair code to an abstract template version). We will not dive into more about that in this work and leave it as future work.

Other findings. We find that on concrete BFPs, all DL-based APR models with pre-training show better repair accuracy results than models trained from scratch (LSTM-based NMT model and Transformer-based NMT model). On abstract BFPs, the trend is almost the same. Particularly, the Transformer-based NMT model (5.61% and 2.00% on BFPs-small and BFPs-medium, respectively) shows better than the LSTM-based NMT model (2.44% and 0.69% on BFPs-small and BFPs-medium, respectively). The reason is apparent: the pre-trained models benefit from the pre-training stage, while models trained from scratch lack this pre-training stage. Additionally, we find all subject DL-based APR models show better repair accuracy results on BFPs-small than BFPs-medium on both abstract BFPs and concrete BFPs. This phenomenon indicates that DL-based APR models perform better on short-sequence code snippets than on long-sequence code snippets. However, from the perspective of CodeBLEU, we find all subject models show higher CodeBLEU results on BFPs-medium than BFPs-small on both abstract BFPs and concrete BFPs. As discussed in section 3.1.4, we use Accuracy@1 as the primary metric for evaluation because it can directly show the percentage of fixed bugs.

Summary for RQ1 and other findings:

(1). Most encoder-decoder-based APR models and the decoder-based APR model have better repair accuracy than encoder-based APR models.

(2). Most DL-based APR models fine-tuned on concrete code datasets have better repair performance than those fine-tuned on abstract code datasets. However, in models trained from scratch, this result is reversed.

(3). All DL-based APR models with pre-training show better repair accuracy results than those trained from scratch.

(4). All subject DL-based models show better repair accuracy results on short-sequence code snippets than long-sequence code snippets.

4.2 RQ2: What Is the Repair Robustness of Different DL-based APR Models against Different Semantic-preserving Code Transformations?

4.2.1 Motivation

Previous works mention that deep neural networks are vulnerable to small perturbations. However, to the best of our knowledge, there needs to be more evaluation on the repair robustness of DL-based APR models against semantic-preserving code transformations. Specifically, we want to quantify APR models' performance change (i.e., robustness) in the repair-specific domain. Furthermore, there are many semantic-preserving code transformations, so we wonder which types significantly impact the repair robustness of DL-based APR models. Lastly, we want to investigate and verify one of our intuitions: whether DL-based APR models with pre-training have better repair robustness than models trained from scratch. If so, we want to quantify this question. This intuition is because pre-trained models (e.g., CodeBERT, GraphCodeBERT, CodeGPT, PLBART, SPT-Code, and CodeT5) have learned general code representation in the pre-training stage, which may build its capability of capturing the code syntax and semantics. In the fine-tuning stage, it further fine-tunes its inner parameters to be adapted to the code repair task. However, the models trained from

scratch (e.g., LSTM-based NMT model and Transformer-based NMT model) are only trained on repair-specific datasets and lack the pre-training stage.

4.2.2 Approach

We conducted the second experiment to answer this research question. Section 3.2 details the experiment setup. We first define nine types of semantic-preserving code transformations, including local variable renaming, method renaming, parameter renaming, boolean exchange, convert switch to if, insert log statement, insert try catch, loop exchange, and reorder condition. Then, we apply each transformation to the test dataset of concrete BFPs. In this experiment, we only employ concrete BFPs instead of abstract BFPs. Another thing is that we find that not all code snippets are suitable for any code transformation. For example, given a method code, if there are no parameters, we cannot apply parameter renaming transformation on it. Considering the motivation of this research question, we want to compare the models' repair performance performed on an original code with that on its transformed code. So when applying each code transformation on the test dataset of concrete BFPs, we first extract the items applicable for this transformation, name the original dataset, and then generate the transformed dataset. Finally, we generate nine original datasets and nine corresponding transformed datasets. Table 3.9 shows the statistics of these datasets.

We employ the fine-tuned models on the previous experiment 3.1. From a high-level perspective, we classify our fine-tuned models as models with pre-training and models trained from scratch. We follow the same hyper-parameter setting for each model as Table 3.1 and 3.2. We evaluate its repair Accuracy@1 and CodeBLEU on each transformed dataset type and its original dataset. Note that we mainly use Accuracy@1 to detail our results because it can accurately present the fixed percentage of bugs. Finally, we summarize our detailed results in Table 4.2 and Table 4.3. Table 4.4 and Table 4.5 summarize the percentage of accuracy reduction.

4.2.3 Results and Discussion

All DL-based APR models meet performance reduction after applying each semantic-preserving transformation. Based on the experiment results in Table 4.2, Table 4.3, and Figure

4.3, we find that all DL-based APR models with pre-training perform worse regarding repair accuracy on transformed datasets than on original datasets for both BFPs-small and BFPs-medium. From the perspective of CodeBLEU, this trend keeps the same as Accuracy@1. This finding verifies that current DL-based APR models exactly suffer from non-robust issues.

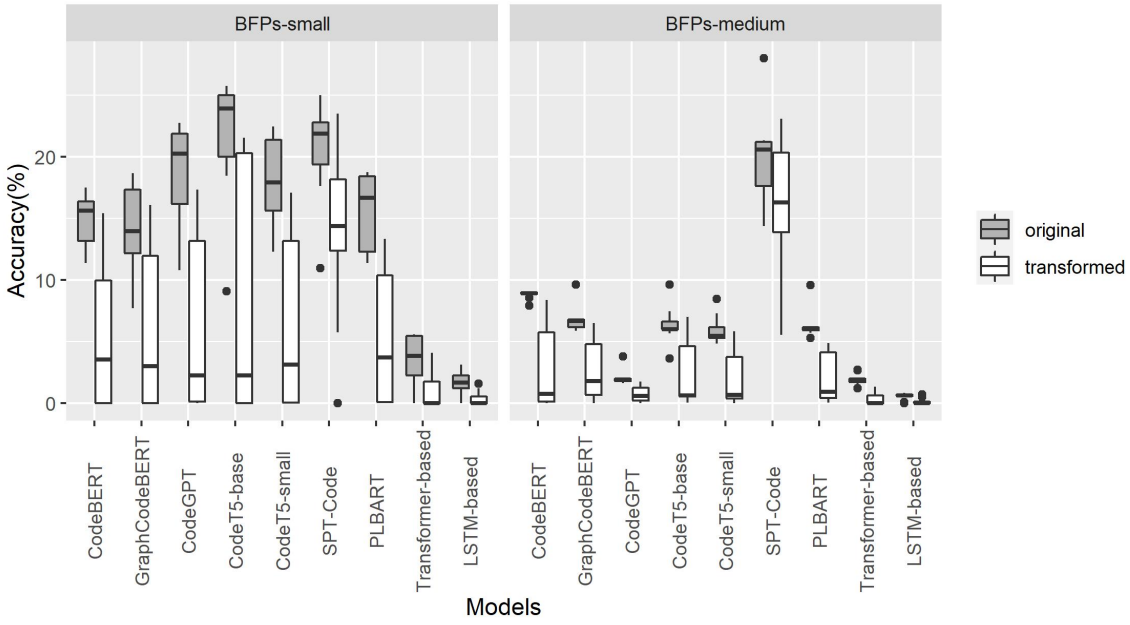

Figure 4.3: The repair accuracy distribution of subject models against different semantic-preserving code transformations.

Renaming-related transformations have a relatively small impact on the repair robustness of DL-based APR models. From Figure 4.4, We find that local variable renaming, method renaming, and parameter renaming have relatively small impacts on models' repair performance among nine transformations. Notably, the SPT-Code shows the best robustness on renaming transformation from Table 4.2. For example, in original and transformed datasets by local variable renaming on BFPs-small, SPT-Code shows 17.02% and 16.27% repair accuracy, respectively; when it comes to method renaming and parameter renaming, SPT-Code shows 20.42% and 20.34% repair accuracy and 21.21% and 20.88% repair accuracy, respectively. Although these three code transformations reduce the repair performance, the pre-trained models show stable robustness.

To explain this phenomenon, we can consider the renaming strategies we present in section 3.2.1. It is because we choose the renaming substitutions based on the masked language prediction

function of the pre-trained model (i.e., GraphCodeBERT), which considers the context of the code snippet. Imagining that if we choose a random and extremely long string for renaming a local variable, it will be tokenized to an entirely different embedding by pre-trained models, which will impact the model's prediction. However, we consider that renaming should comply with standard coding conventions in the real world and consider the natural context of token-surrounded code snippets.

Transformations related to changing the syntactic structure of code have a relatively significant impact on the repair robustness of DL-based APR models. From Figure 4.4, we find that on BFPs-small, insert a log statement, insert try catch, boolean exchange, and loop exchange have a more significant impact on the model's repair performance. Particularly, loop exchange performs all 0% (Table 4.2) repair accuracy on the transformed datasets. While recorder condition and convert switch to if are a little better than the former four, they also introduce a big robustness problem of these pre-trained models. However, we also find that on BFPs-small, SPT-Code shows relatively better robustness than other models, particularly in convert switch to if transformation, which is 20.45% and 18.18% on the original dataset and transformed dataset, respectively.

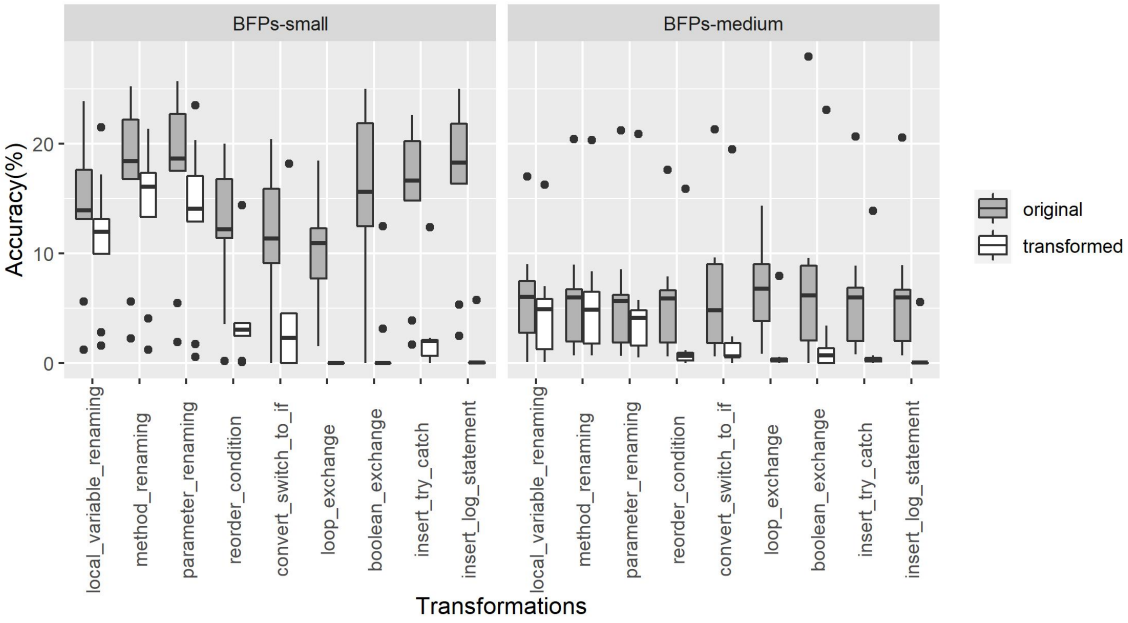

Figure 4.4: The repair accuracy distribution of semantic-preserving code transformations on subject models.

DL-based APR models trained from scratch show more significant non-robust issues than models with pre-training. From Table 4.2, 4.3, 4.4 and 4.5, we find the LSTM-based NMT model and the Transformer-based NMT model keep a lower repair accuracy no matter what on the original datasets and transformed dataset. We cannot compare them with other pre-trained models by performance reduction only because if they perform the same 0% and 0% on both original and transformed datasets, we cannot conclude they have stable robustness. However, in high-level and whole-data analysis, DL-based APR models trained from scratch show worse repair robustness than pre-trained APR models. For example, for insert a log statement, insert try catch, convert switch to if, boolean exchange, and loop exchange, both the LSTM-based NMT model and the Transformer-based NMT model perform 0% repair accuracy on the corresponding transformed datasets. The reason behind this is apparent, the pre-trained models benefit from the pre-training stage for building an initial general capability of code understanding and generation, while APR models trained from scratch lack this stage.

Summary for RQ2 and other findings:

(1). All DL-based APR models meet performance reduction (i.e., robustness) after applying a semantic-preserving transformation.

(2). Renaming-related transformations (i.e., local variable renaming, method renaming, and parameter renaming) have a relatively small impact on the repair robustness of DL-based APR models.

(3). Transformations related to changing the syntactic structure of code have a relatively significant impact on the repair robustness of DL-based APR models.

(4). DL-based APR Models trained from scratch show more considerable performance reduction (i.e., robustness) than models with pre-training.

Table 4.2: Fine-tuned models against different semantic-preserving code transformations on BFPs-small. "acc" stands for Accuracy@1(%) and "codebleu" stands for CodeBLEU(%).

	local_variable_renaming				**method_renaming**				**parameter_renaming**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	13.15	80.60	9.96	79.76	16.76	80.56	15.42	80.09	17.51	81.55	13.12	80.39
GraphCodeBert	13.94	80.58	11.95	79.94	17.68	79.62	16.09	79.26	18.67	80.67	14.05	79.50
CodeGPT	17.53	79.37	13.15	79.79	22.21	79.70	17.34	79.09	22.74	80.70	16.43	79.58
CodeT5-base	23.90	82.40	21.51	82.98	25.24	80.76	20.29	80.47	25.73	81.59	20.33	80.43
CodeT5-small	17.93	80.70	13.15	80.38	21.80	79.28	16.95	79.15	22.44	80.22	17.07	79.71
SPT-Code	17.60	82.21	17.20	81.87	22.77	83.48	21.38	83.40	24.98	85.17	23.50	84.79
PLBART	13.15	81.34	10.36	80.92	18.40	77.64	13.31	77.79	18.41	79.33	12.92	79.21
LSTM-based	1.20	49.63	1.59	44.14	2.24	49.87	1.23	49.86	1.92	51.98	0.55	44.41
Transformer-based	5.58	59.84	2.79	56.35	5.59	62.60	4.07	60.88	5.46	63.49	1.74	55.04
	boolean_exchange				**loop_exchange**				**reorder_condition**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	15.62	80.25	0.00	78.91	12.31	76.78	0.00	69.28	11.38	79.10	3.56	74.10
GraphCodeBert	12.50	84.45	0.00	77.71	7.69	82.78	0.00	69.80	12.18	79.36	3.02	74.21
CodeGPT	21.88	81.80	0.00	68.89	10.77	80.39	0.00	68.57	16.18	77.66	2.49	70.67
CodeT5-base	25.00	86.04	0.00	75.49	18.46	84.85	0.00	70.18	20.00	79.73	2.93	72.08
CodeT5-small	15.63	80.51	3.13	75.02	12.31	82.80	0.00	74.82	16.80	77.89	3.64	72.31
SPT-Code	21.88	85.80	12.50	84.97	10.94	80.20	0.00	74.35	19.38	83.48	14.38	81.57
PLBART	18.75	80.75	0.00	77.00	12.31	80.88	0.00	72.60	11.56	74.89	3.73	71.87
LSTM-based	3.12	45.66	0.00	39.63	1.54	55.77	0.00	27.13	0.18	51.80	0.09	41.57
Transformer-based	0.00	55.97	0.00	34.06	1.54	66.87	0.00	34.45	3.56	61.39	0.18	41.73
	convert_switch_to_if				**insert_log_statement**				**insert_try_catch**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	15.91	83.58	4.55	76.42	16.37	80.46	0.00	68.04	14.81	80.92	1.75	74.18
GraphCodeBert	9.09	81.02	4.55	74.50	17.33	79.59	0.02	67.64	16.11	79.66	2.08	71.35
CodeGPT	15.91	87.10	2.27	73.77	21.81	79.53	0.14	66.60	20.23	79.68	2.26	69.42
CodeT5-base	9.09	84.14	2.27	76.71	25.00	80.83	0.00	67.65	22.61	80.99	2.02	67.20
CodeT5-small	13.64	85.39	0.00	73.61	21.38	79.33	0.05	68.65	19.69	79.54	2.08	67.26
SPT-Code	20.45	86.68	18.18	81.83	22.91	83.43	5.75	75.24	22.65	83.37	12.39	78.23
PLBART	11.36	80.15	4.55	69.77	18.26	77.52	0.07	68.77	16.65	77.09	0.63	64.17
LSTM-based	0.00	56.09	0.00	39.41	2.48	49.58	0.00	34.31	1.69	46.61	0.00	33.01
Transformer-based	2.27	68.64	0.00	46.61	5.33	62.56	0.00	38.21	3.85	62.20	0.00	40.03

Table 4.3: Fine-tuned models against different semantic-preserving code transformations on BFPs-medium. "acc" stands for Accuracy@1(%) and "codebleu" stands for CodeBLEU(%).

	local_variable_renaming				**method_renaming**				**parameter_renaming**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	9.01	87.12	5.82	86.88	8.96	87.68	8.38	87.63	8.53	87.67	5.74	87.23
GraphCodeBert	6.01	86.25	4.91	86.33	6.74	87.13	6.50	87.19	6.21	87.36	4.81	86.95
CodeGPT	1.64	87.04	1.27	86.98	1.91	87.75	1.75	87.73	1.81	87.96	1.56	87.33
CodeT5-base	7.46	76.26	7.01	75.78	6.00	72.08	4.92	71.80	5.65	71.52	4.63	69.66
CodeT5-small	7.28	75.85	5.82	75.39	5.47	71.50	4.55	71.29	5.14	71.00	3.74	69.09
SPT-Code	17.02	89.17	16.27	89.11	20.42	90.29	20.34	90.24	21.21	90.51	20.88	90.23
PLBART	5.91	86.67	4.19	86.33	6.10	87.07	4.86	87.01	5.70	87.55	4.11	87.28
LSTM-based	0.09	76.86	0.09	73.76	0.71	77.33	0.71	77.10	0.63	77.89	0.51	74.70
Transformer-based	2.73	72.80	0.64	69.98	1.96	73.78	1.35	73.15	1.88	74.04	0.79	69.04
	boolean_exchange				**loop_exchange**				**reorder_condition**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	8.90	88.88	0.00	77.57	9.04	87.29	0.14	76.43	7.91	86.99	0.75	79.22
GraphCodeBert	6.16	88.39	3.42	79.39	6.78	86.30	0.28	76.96	5.87	86.40	1.15	80.37
CodeGPT	2.05	87.07	0.68	79.51	3.81	87.79	0.28	77.55	1.87	87.43	0.22	80.50
CodeT5-base	6.16	72.35	0.68	66.38	9.60	79.59	0.57	70.88	6.62	74.26	0.64	69.21
CodeT5-small	6.16	71.55	0.68	65.93	8.47	78.73	0.28	71.25	5.95	73.61	0.64	69.62
SPT-Code	27.97	90.89	23.08	90.53	14.37	88.61	7.92	87.00	17.61	89.86	15.89	89.30
PLBART	9.59	88.03	1.37	82.84	6.07	86.62	0.42	81.85	5.28	86.24	0.92	82.92
LSTM-based	0.00	78.83	0.00	64.35	0.85	78.28	0.00	52.73	0.61	76.95	0.11	64.02
Transformer-based	1.37	73.26	0.00	46.95	2.68	75.80	0.00	45.13	1.73	73.08	0.00	48.07
	convert_switch_to_if				**insert_log_statement**				**insert_try_catch**			
	original		transformed		original		transformed		original		transformed	
	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu	acc	codebleu
CodeBERT	9.04	89.68	2.41	78.11	8.91	87.70	0.02	76.00	8.89	87.63	0.47	79.36
GraphCodeBert	9.64	89.53	1.81	76.09	6.67	87.17	0.00	76.08	6.88	87.04	0.67	79.51
CodeGPT	1.81	90.23	0.60	75.90	1.93	87.75	0.00	74.79	2.01	87.73	0.20	80.64
CodeT5-base	3.61	72.75	0.60	65.98	6.00	71.86	0.03	63.92	5.99	71.99	0.24	61.12
CodeT5-small	4.82	72.09	1.81	66.26	5.44	71.26	0.02	64.42	5.28	71.35	0.37	62.02
SPT-Code	21.34	90.93	19.51	89.07	20.56	90.31	5.56	85.46	20.65	90.37	13.86	87.76
PLBART	6.02	88.97	0.60	84.02	6.16	87.14	0.03	80.31	6.21	86.98	0.14	81.63
LSTM-based	0.60	76.14	0.00	60.75	0.70	77.29	0.00	62.30	0.77	77.07	0.00	61.75
Transformer-based	1.20	74.34	0.00	48.67	1.99	73.82	0.00	46.57	1.89	73.40	0.00	47.18

Table 4.4: Accuracy reduction on BFPs-small (%).

	local_variable_renaming	**method_renaming**	**parameter_renaming**
CodeBERT	24.26	8.00	25.07
GraphCodeBert	14.28	8.99	24.75
CodeGPT	24.99	21.93	27.75
CodeT5-base	10.00	19.62	20.99
CodeT5-small	26.67	22.23	23.93
SPT-Code	2.27	6.10	5.92
PLBART	21.22	27.66	29.82
LSTM-based	-32.50	45.09	71.35
Transformer-based	50.00	27.19	68.13
	boolean_exchange	**loop_exchange**	**reorder_condition**
CodeBERT	100.00	100.00	68.72
GraphCodeBert	100.00	100.00	75.21
CodeGPT	100.00	100.00	84.61
CodeT5-base	100.00	100.00	85.33
CodeT5-small	80.00	100.00	78.31
SPT-Code	42.87	100.00	25.80
PLBART	100.00	100.00	67.73
LSTM-based	100.00	100.00	50.00
Transformer-based	0.00	100.00	94.94
	convert_switch_to_if	**insert_log_statement**	**insert_try_catch**
CodeBERT	71.40	100.00	88.18
GraphCodeBert	49.94	99.88	87.09
CodeGPT	85.73	99.36	88.83
CodeT5-base	75.00	100.00	91.08
CodeT5-small	100.00	99.75	89.45
SPT-Code	11.10	74.90	45.30
PLBART	59.95	99.62	96.22
LSTM-based	0.00	100.00	100.00
Transformer-based	100.00	100.00	100.00

Table 4.5: Accuracy reduction on BFPs-medium (%).

	local_variable_renaming	**method_renaming**	**parameter_renaming**
CodeBERT	35.41	6.47	32.71
GraphCodeBert	18.30	3.56	22.54
CodeGPT	22.56	8.38	13.81
CodeT5-base	6.10	17.96	18.11
CodeT5-small	20.00	16.77	27.15
SPT-Code	4.41	0.39	1.56
PLBART	29.10	20.33	27.89
LSTM-based	0.00	0.00	19.05
Transformer-based	76.56	31.12	57.98
	boolean_exchange	**loop_exchange**	**reorder_condition**
CodeBERT	100.00	98.45	90.52
GraphCodeBert	44.48	95.87	80.41
CodeGPT	66.83	92.65	88.24
CodeT5-base	88.89	94.12	90.30
CodeT5-small	88.89	96.67	89.20
SPT-Code	17.48	44.89	9.77
PLBART	85.71	93.08	82.58
LSTM-based	0.00	100.00	81.97
Transformer-based	100.00	100.00	100.00
	convert_switch_to_if	**insert_log_statement**	**insert_try_catch**
CodeBERT	73.34	99.78	94.71
GraphCodeBert	81.22	100.00	90.26
CodeGPT	66.85	100.00	90.05
CodeT5-base	83.33	99.48	96.05
CodeT5-small	62.50	99.71	92.91
SPT-Code	8.58	72.96	32.88
PLBART	90.03	99.51	97.75
LSTM-based	100.00	100.00	100.00
Transformer-based	100.00	100.00	100.00

Chapter 5

Threats To Validity

In this chapter, we discuss the threats to validity of this thesis.

5.1 Internal Validity

For fine-tuning pre-trained models on RQ1, an essential factor that may impact the repair result is the hyper-parameter setting. Any changes, such as learning rate and training batch size, may impact the fine-tuned model slightly. We use the same hyper-parameter settings as in previous works to experiment fairly. However, due to the limit of GPU memory, we meet problems such as "CUDA out of memory" in fine-tuning. It is because the loaded data and the vast number of model parameters occupy too much memory of the GPU. In such cases, we reduce the training batch size, e.g., reducing the batch size from 64 to 32 for runnable fine-tuning. We also meet gradients exploring problems, and we reduce the initial learning rate gradually and slightly for runnable fine-tuning. Finally, Table 3.3 and 3.4 list our hyper-parameter settings. At a high level, we keep most of the same settings with previous works to avoid potential biases.

For substitutions generated for renaming local variable names, methods, and parameters on RQ2, we only use pre-trained GraphCodeBERT's masked language prediction function to generate the substitutions. However, other pre-trained models, such as CodeT5 and SPT-Code, also have alternative functions to predict masked tokens. In particular, CodeT5 designs a masked identifier prediction objective, and SPT-Code designs a method name generation objective in their pre-training

stage, which is helpful for us to generate renaming substitutions. However, most of these pre-trained models rely on the same pre-training dataset, such as CodeSearchNet (Husain et al., 2019), so predicted masked tokens by different pre-trained models may be closed in the semantic context.

For the semantic-preserving transformations we chose in the section 3.2.1, some of them may be rejected by compilers and developers in practice. For example, the original code should have a throw in the method declaration when we insert a try-catch block to a method-level code snippet. However, as we perform code transformations for robustness testing instead of code refactoring in this paper, we consider that such biases should be minor. We leave how code refactoring impacts the performance of DL-based APR models as future work.

5.2 External Validity

Although we chose nine models as subject models, our experiment result may not be generalized to other latest models. As in the field of code intelligence, various powerful large code models have been designed in recent three years. Even with the same training dataset and same model architecture, the new-designed and different pre-training objectives may significantly impact the model's result. Therefore, we try to choose SOTAs and representative models to increase the generalization of our experiment results. Finally, three types of popular pre-trained models are included in our subject models.

Additionally, we define nine semantic-preserving code transformations in RQ2, but their result on different models cannot represent other undefined types of transformations. Therefore, we leave this as future work.

5.3 Construct Validity

We choose Accuracy@1 (i.e., top one predicted code) as our metric to evaluate whether the generated fixed code is the same as expected. Although almost all recent works utilize this metric for code repair evaluation, it may introduce biases. For example, even if the generated fixed code differs from the expected fixed code in syntactic structure, it may still be the correct one. However, in the field of DL-based APR, most models do not employ test suites to validate the generated fixed

code candidates due to the lack of test cases. To relieve this problem, we introduce the CodeBLEU as the supplementary metric for evaluation.

Chapter 6

Conclusion and Future Work

This chapter summarizes our insights and contributions based on our experiments conducted in this thesis. Additionally, we present the potential future works that may enhance the experiments in this thesis for better DL-based APR approaches.

6.1 Summary of the Thesis

In this thesis, we mainly investigate the repair performance of current DL-based APR models and the repair robustness of DL-based APR models against semantic-preserving code transformations. We design two experiments to answer the corresponding two research questions. The first experiment is to fine-tune all subject models on unified datasets for a thorough and fair comparison of the repair performance. Further, the second experiment employs the fine-tuned models in the first experiment to evaluate their robustness against nine semantic-preserving code transformations. In the first experiment, our main findings are: (1). Most encoder-decoder-based APR models and the decoder-based APR model have better repair accuracy than encoder-based APR models; (2). Most DL-based APR models fine-tuned on the concrete code datasets have better repair performance than those fine-tuned on the abstract code dataset; (3). All subject DL-based APR models with pre-training show better repair accuracy results than models trained from scratch. Our main findings in the second experiment are: (4). DL-based APR models suffer from non-robust issues; (5). The

semantic-preserving transformations related to the change of syntactic structure have a more significant impact on the repair robustness of DL-based APR models compared with renaming-related transformations; (6). DL-based APR models trained from scratch show more significant non-robust issues (i.e., robustness) than models with pre-training.

Our findings provide insights for the following research: (1). Encoder-decoder architecture is a better choice for designing program repair models; (2). The concrete BFPs benchmark is a better choice for evaluating the program repair capability of DL-based APR models instead of abstract BFPs; (3). The paradigm of pre-training and fine-tuning is better than the paradigm of training from scratch on the robustness of DL-based APR models; (4). Investigation about how to effectively apply semantic-preserving code transformations to boost the repair performance and robustness of DL-based APR models is needed.

6.2 Future Work

One of the contributions of this thesis is to quantify the repair robustness of DL-based APR models against semantic-preserving transformations. However, there is an urgent lack of work on a better benchmark dataset, a full-automatic code refactoring tool, and how to effectively apply semantic-preserving code transformations to boost the models' repair performance and robustness. Lastly, we leave the investigation of how code refactoring impacts the robustness of DL-based APR models as future work.

6.2.1 Constructing a Better Benchmark Dataset for DL-based APR Approaches

The datasets we used are from (Tufano et al., 2019) published in 2019, and the maximum length of each code case is lower than 100. However, different DL-based APR models are developed and released quickly. These datasets may not satisfy current evaluation requirements for repair performance and robustness of DL-based APR models. Firstly, the current dataset only offers one fixed code for each buggy code. Most of the previous work used Accuracy@1 as the primary metric, so any correct candidates with different formats are judged as incorrect. Therefore, it is necessary to introduce more semantic-preserving code transformations for the expected fixed code

to solve this problem or to introduce test cases for evaluating the correctness of predicted fixed code. Additionally, the length of code snippets can be expanded because of the improvement of DL-based APR models. Furthermore, small code snippets exactly restrict the applicable semantic-preserving code transformations, which further limits the robustness evaluation of DL-based APR models.

6.2.2 Designing a Full-automatic Code Transformations Tool for Partial Code Snippet

When we perform the semantic-preserving code transformations on experiments, we find most of the current refactoring tools are semi-automatic, e.g., users need to provide a method name for renaming a method. Furthermore, they are designed for complete code, e.g., project-level or java-file level, instead of the partial code snippet. However, it is necessary to design a full-automatic code transformations tool for partial code snippets in the fields such as robustness testing and adversarial attack for code models. Such a tool will definitely boost the research of the DL-based APR community.

6.2.3 Investigating Effectively Apply Code Transformations on Training Dataset Augmentation for Better DL-based APR Approaches

Fine-tuning a large pre-trained model on current datasets may need several days for one GPU with 32 GB memory. However, when we want to apply many semantic-preserving code transformations on a training dataset for data augmentation, it may take several weeks on our experiment device. Considering our GPU resource limitation, we leave this as future work. However, how to effectively apply code transformations on training dataset augmentation for better DL-based APR models is still needed. For example, how to use a relatively small training dataset but offer rich diversity for effectively training a better DL-based APR model.

References

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Apr-models-performance. (n.d.). https://github.com/ThomasShiyu/APR-Models-Performance.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Berabi, B., He, J., Raychev, V., & Vechev, M. (2021). Tfix: Learning to fix coding errors with a text-to-text transformer. In *International conference on machine learning* (pp. 780–791).

Bielik, P., & Vechev, M. (2020). Adversarial robustness for code. In *International conference on machine learning* (pp. 896–907).

Bigquery. (n.d.). https://console.cloud.google.com/marketplace/details/github/github-repos.

Cambronero, J., Li, H., Kim, S., Sen, K., & Chandra, S. (2019). When deep learning met code search. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 964–974).

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P., & Ray, B. (2022). Natgen: Generative pre-training by" naturalizing" source code. *arXiv preprint arXiv:2206.07585*.

Chakraborty, S., Ding, Y., Allamanis, M., & Ray, B. (2020). Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*.

Chakraborty, S., & Ray, B. (2021). On multi-modal learning of editing source code. In *2021 36th ieee/acm international conference on automated software engineering (ase)* (pp. 443–455).

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., ... others (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., & Monperrus, M. (2019). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, *47*(9), 1943–1959.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ding, Z., Li, H., & Shang, W. (2022). Logentext: Automatically generating logging texts using neural machine translation. *SANER. IEEE*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... others (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Gao, X., Mechtaev, S., & Roychoudhury, A. (2019). Crash-avoiding program repair. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis* (pp. 8–18).

Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T., & Mkaouer, M. W. (2021). One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 1303–1313).

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... others (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *Thirty-first aaai conference on artificial intelligence.*

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2018). Deep code comment generation. In *2018 ieee/acm 26th international conference on program comprehension (icpc)* (pp. 200–20010).

Hua, J., Zhang, M., Wang, K., & Khurshid, S. (2018). Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 888–891).

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., & Chen, X. (2018). Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis* (pp. 298–309).

Jiang, N., Lutellier, T., & Tan, L. (2021). Cure: Code-aware neural machine translation for automatic program repair. In *2021 ieee/acm 43rd international conference on software engineering (icse)* (pp. 1161–1173).

Just, R., Jalali, D., & Ernst, M. D. (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* (pp. 437–440).

Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (icse)* (pp. 802–811).

Le, X. B. D., Lo, D., & Le Goues, C. (2016a). History driven program repair. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, p. 213-224). doi: 10.1109/SANER.2016.76

Le, X. B. D., Lo, D., & Le Goues, C. (2016b). History driven program repair. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, pp. 213–224).

Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th international conference on software engineering (icse)* (pp. 3–13).

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, *38*(1), 54-72. doi: 10.1109/TSE.2011.104

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ... Zettlemoyer, L. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Li, J., Wang, Y., Lyu, M. R., & King, I. (2017). Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Liang, J., Ji, R., Jiang, J., Zhou, S., Lou, Y., Xiong, Y., & Huang, G. (2021). Interactive patch filtering as debugging aid. In *2021 ieee international conference on software maintenance and evolution (icsme)* (pp. 239–250).

Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings companion of the 2017 acm sigplan international conference on systems, programming, languages, and applications: software for humanity* (pp. 55–56).

Liu, C., Yang, J., Tan, L., & Hafiz, M. (2013). R2fix: Automatically generating bug fixes from bug reports. In *2013 ieee sixth international conference on software testing, verification and validation* (pp. 282–291).

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Long, F., & Rinard, M. (2015). Staged program repair with condition synthesis. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 166–178).

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., ... others (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., & Tan, L. (2020). Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th acm sigsoft international symposium on software testing and analysis* (pp. 101–114).

Mechtaev, S., Yi, J., & Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 1, pp. 448–458).

Müller, R., Kornblith, S., & Hinton, G. E. (2019). When does label smoothing help? *Advances in neural information processing systems*, *32*.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., & Dig, D. (2013). A comparative study of manual and automated refactorings. In *European conference on object-oriented programming* (pp. 552–576).

Negara, S., Vakilian, M., Chen, N., Johnson, R. E., & Dig, D. (2012). Is it dangerous to use version

control histories to study source code evolution? In *European conference on object-oriented programming* (pp. 79–103).

Niu, C., Li, C., Ng, V., Ge, J., Huang, L., & Luo, B. (2022). Spt-code: sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th international conference on software engineering* (pp. 2006–2018).

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the association for computational linguistics* (pp. 311–318).

Pour, M. V., Li, Z., Ma, L., & Hemmati, H. (2021). A search-based testing framework for deep neural networks of source code embedding. In *2021 14th ieee conference on software testing, verification and validation (icst)* (pp. 36–46).

Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 international symposium on software testing and analysis* (pp. 24–36).

Rabin, M. R. I., Bui, N. D., Wang, K., Yu, Y., Jiang, L., & Alipour, M. A. (2021). On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, *135*, 106552.

Rabin, M. R. I., Wang, K., & Alipour, M. A. (2019). Testing neural program analyzers. *arXiv preprint arXiv:1908.10711*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, *1*(8), 9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... others (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, *21*(140), 1–67.

Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S., & Reps, T. (2020). Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., ... Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Saha, R. K., Lyu, Y., Yoshida, H., & Prasad, M. R. (2017). Elixir: Effective object-oriented program

repair. In *2017 32nd ieee/acm international conference on automated software engineering (ase)* (pp. 648–659).

tree-sitter. (n.d.). https://github.com/tree-sitter/tree-sitter.

Tsantalis, N., Ketkar, A., & Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*.

Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., & Poshyvanyk, D. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *28*(4), 1–29.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., & Yu, P. S. (2018). Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (pp. 397–407).

Wang, D., Jia, Z., Li, S., Yu, Y., Xiong, Y., Dong, W., & Liao, X. (2022). Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th international conference on software engineering* (pp. 287–298).

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Wei, H., & Li, M. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Ijcai* (pp. 3034–3040).

Wei, M., Huang, Y., Yang, J., Wang, J., & Wang, S. (2021). Cocofuzzing: Testing neural code models with coverage-guided fuzzing. *arXiv preprint arXiv:2106.09242*.

Wen, M., Chen, J., Wu, R., Hao, D., & Cheung, S.-C. (2018). Context-aware patch generation for better automated program repair. In *2018 ieee/acm 40th international conference on software engineering (icse)* (pp. 1–11).

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. L., Durieux, T., ... Monperrus, M. (2016). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, *43*(1), 34–55.

Yang, B., & Yang, J. (2020). Exploring the differences between plausible and correct patches at fine-grained level. In *2020 ieee 2nd international workshop on intelligent bug fixing (ibf)* (pp. 1–8).

Yang, J., Tan, L., Peyton, J., & Duer, K. A. (2019). Towards better utilizing static application security testing. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (pp. 51–60).

Yang, J., Zhikhartsev, A., Liu, Y., & Tan, L. (2017). Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 831–841).

Yang, Z., Shi, J., He, J., & Lo, D. (2022). Natural attack for pre-trained models of code. *arXiv preprint arXiv:2201.08698*.

Ye, H., Martinez, M., & Monperrus, M. (2022). Neural program repair with execution-based back-propagation. In *Proceedings of the 44th international conference on software engineering* (pp. 1506–1518).

Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., & Jin, Z. (2020). Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 34, pp. 1169–1176).

Zhang, J., Panthaplackel, S., Nie, P., Li, J. J., & Gligoric, M. (2022). Coditt5: Pretraining for source code and natural language editing. *arXiv preprint arXiv:2208.05446*.

Zhang, W. E., Sheng, Q. Z., Alhazmi, A., & Li, C. (2020). Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *11*(3), 1–41.

Zhang, Z., Zhang, H., Shen, B., & Gu, X. (2022). Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th acm joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1073–1084).

Zhu, Q., Sun, Z., Xiao, Y.-a., Zhang, W., Yuan, K., Xiong, Y., & Zhang, L. (2021). A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 341–353).