Synthetic Execution Data for Program Repair Instruction Tuning

André Silva
KTH Royal Institute of Technology
Stockholm, Sweden
andreans@kth.se

ABSTRACT

Fine-tuning large language models (LLMs) for program repair is a promising area of research. However, collecting high-quality, real-world data for this task is challenging and costly. Synthetic data generation offers a potential solution to this problem. In this work, we study the use of synthetic execution data in fine-tuning LLMs for program repair. We generate synthetic test cases and error messages to augment existing bug-fixing datasets. Our experiments show that fine-tuning with this synthetic data improves pass@1 scores on two benchmarks: Defects4J and GitBug-Java. To the best of our knowledge, we are the first to explore synthetic execution data for program repair fine-tuning.

ACM Reference Format:
André Silva. 2024. Synthetic Execution Data for Program Repair Instruction Tuning. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fine-tuning is a prominent area of program repair [15] where large language models (LLMs) are fine-tuned with datasets of bug-fixing commits to learn the task of repairing bugs. However, fine-tuning typically relies on data collected from the real world, either from the web or from hiring human annotators to create such data. Collecting such data entails significant costs and is inherently limited in scale.

To overcome this challenge, synthetic data is often generated and used to train models, with several fine-tuning datasets being developed, primarily by employing a powerful large language model such as GPT4 to generate high-quality samples [5, 18, 19, 21]. In program repair, existing work focuses on synthetic bugs, where incorrect code is inserted in existing programs [2, 6, 16, 22, 23].

In this work, we study synthetic execution data. Execution data is fundamental in program repair: test cases define the expected behavior of the program under test, and failing assertions expose existing bugs. However, obtaining real execution data is hard since it involves executing software in the wild. To overcome this challenge, we generate synthetic execution data (i.e., a failing test case and error) for existing real-world bugs.

Our experiments show that fine-tuning an LLM with synthetic execution data improves pass@1 scores over two benchmarks, Defects4J [9] and GitBug-Java [17], against the non-fine-tuned baseline. These results indicate that the synthetic execution data enables the model to use the information from real execution data at inference time. We make our dataset, model, and code available online.

In summary, our contributions are:

- We study synthetic execution data for fine-tuning LLMs for program repair. Our results show that synthetic execution data enables using real execution data at inference time. To the best of our knowledge, we are the first to explore synthetic execution data.
- We make our research artifacts available online at https://t.ly/rambv.

2 MOTIVATION

Real Buggy Function:

```
Message json() {
    return new Message("Hello, world");
}
```

Real Fixed Function:

```
Message json() {
    return new Message("Hello, World!");
}
```

Synthetic Failing Test:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MessageTest {
    @Test
    public void testJsonReturnsCorrectMessage() {
        Message message = new Message();
        assertEquals("Hello, World!",
             message.json().getContent());
    }
}
```

Synthetic Error:

```
java.lang.AssertionError: expected:<Hello, World!>
     but was:<Hello, world>
 at org.junit.Assert.fail(Assert.java:88)
 at org.junit.Assert.assertEquals(Assert.java:119)
 at org.junit.Assert.assertEquals(Assert.java:146)
 at MessageTest.testJsonReturnsCorrectMessage
      (MessageTest.java:7)
```

Figure 1: Example of a real bug-fixing commit enriched with synthetic execution data. Without access to the real fixed function, it is impossible to infer the expected behavior of the buggy function. Our approach generates execution data exposing the difference between expected and actual behavior.

Conference'17, July 2017, Washington, DC, USA

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

Software bugs exist when the behavior of a program is different from expected. In software, the expected behavior is usually given by test cases, where programs are tested against inputs and expected outputs. For example, Figure 1 shows an example of a real bug where the buggy function returns a message containing the string *Hello, world*. This function is buggy because the expected behavior, which can be inferred from the answer, is to uppercase the word *World* and add an exclamation mark at the end of the string. However, if only provided with the buggy function, an LLM cannot be certain of what the expected behavior is since it lacks the information provided by the test case.

While fine-tuning LLMs for program repair improves bug-fixing abilities, fine-tuning with expected behavior is underexplored by the literature. We argue that this is because obtaining samples containing the expected behavior is not straightforward: 1) test execution runs are not easily available, and 2) reproducing test execution runs is a hard task at scale [14].

Our intuition is that existing LLMs can be guided to generate useful synthetic data to enrich existing bug-fixing datasets. In the same example (Figure 1), a synthetic failing test and a synthetic error, generated by *gpt-4o-mini*, are shown. Here, the expected and current behavior is clear: the test case asserts if the returned message equals the expected one, and the test error highlights the difference between observed and expected behavior.

3 APPROACH

3.1 Generating Synthetic Execution Data

Generating synthetic data with large language models (LLMs) requires careful consideration of both the input data and the prompts used to guide the model's generation process. The generation process starts with high-quality seeds — existing code snippets, bug-fixing diffs, or other relevant but incomplete data for the downstream tasks we want to fine-tune a model for. In our case, program repair, these seeds come from curated bug-fixing commits, which ground the generation task to specific bug-fixing changes in real codebases.

Alongside these seeds, well-crafted instructions are critical to ensure the generated data is aligned with expected model use cases. When using a model to generate bug-fixing patches, developers typically provide information about failing test cases related to the bug they are trying to fix. Given the seed bug-fixing commit, we thus prompt the model to generate a difference-exposing test case [7] (i.e., a test that exposes different behavior due to the changes made in the code) and the associated failure message (e.g., an assertion or runtime error).

By systematically generating synthetic execution data, we aim to create a rich dataset that can be used to fine-tune program repair models. The combination of bug fixes, test cases, and runtime information provides a comprehensive training set that can improve the model's ability to handle real-world repair tasks, without requiring large amounts of curated data.

3.2 Instruction Tuning for Program Repair

Instruction tuning is a fine-tuning step that adapts LLMs to follow natural language instructions. By enhancing the model's ability to understand and follow task-specific instructions, instruction tuning can improve its performance on downstream tasks. Instruction tuning for program repair serves two main purposes: 1) It helps the model learn the task of program repair beyond merely mimicking examples provided in the prompt (e.g., in few-shot prompting) or

Prompt:

````
You are an automatic program repair tool. Your task is to
     fix the provided buggy code.
The following code contains a buggy function:
```java
    /**
     * Puts all values of this record into the given Map.
     *
     * @param map The Map to populate.
     * @return the given map.
     */
    <M extends Map<String, String>> M putIn(final M map) {
        for (final Entry<String, Integer> entry :
             mapping.entrySet()) {
            final int col = entry.getValue().intValue();
                map.put(entry.getKey(), values[col]);
        }
        return map;
    }
```

The code fails the following test:
```java
    @Test
    public void testToMapWithShortRecord() throws Exception
         {
       final CSVParser parser =  CSVParser.parse("a,b",
            CSVFormat.DEFAULT.withHeader("A", "B", "C"));
       final CSVRecord shortRec = parser.iterator().next();
       shortRec.toMap();
    }
```

With the following test error:
```
java.lang.ArrayIndexOutOfBoundsException: 2
```

Please provide a fixed version of the buggy function, and
     only that function:
````

Generated Patch:

```
@@ -1,13 +1,15 @@
     /**
      * Puts all values of this record into the given Map.
      *
      * @param map The Map to populate.
      * @return the given map.
      */
     <M extends Map<String, String>> M putIn(final M map) {
         for (final Entry<String, Integer> entry :
              mapping.entrySet()) {
             final int col = entry.getValue().intValue();
+            if (col < values.length) {
                 map.put(entry.getKey(), values[col]);
             }
+        }
         return map;
     }
```

Figure 2: Prompt and exact-match patch for bug Csv-6 of Defects4J, exclusively generated by the fine-tuned model. The test case/error information, indicating an out-of-bounds bug, guides the model to generate the correct patch.

Table 1: pass@1 of both models on Defects4J and GitBug-Java

Model	Defects4J v2 (477 bugs)			GitBug-Java (49 bugs)		
	plausible@1	ast-match@1	exact-match@1	plausible@1	ast-match@1	exact-match@1
CodeLlama-7b-Instruct	2.4%	0.5%	0.4%	3.3%	1.4%	1.4%
CodeLlama-7b-Instruct-ft	**5.4%**	**2.0%**	**1.8%**	**4.5%**	**1.6%**	**1.6%**

following unseen instructions. This involves understanding the structure of code, identifying potential bugs, and generating appropriate fixes. 2) It enables the model to effectively utilize the information available in a repair setting, such as failing test cases and error messages. This contextual information is crucial in understanding expected and current behavior, to generate correct patches.

We construct an instruction-tuning dataset specifically designed for program repair tasks. Each entry in this dataset consists of a prompt and an answer. The prompt is carefully crafted to include three key elements: a) The buggy function that needs to be repaired. b) A synthetically generated test case that exposes the bug. c) An error message or stack trace associated with the failing test. The answer component of each dataset entry contains the correctly fixed version of the function.

4 EXPERIMENTAL METHODOLOGY

Model We choose to fine-tune CodeLlama-7b-Instruct[1] [13], a publicly available LLM released in 2023 and trained on 500B code tokens, following the criteria of RepairLLaMA [15]. This model is particularly suitable for our study as it has been trained for code-related tasks and instruction following.

Dataset We select the single-function subset of MegaDiff [11] as the seed dataset, which contains bug-fixing commits focused on individual functions or methods, and process it as follows. First, we remove duplicate samples from the dataset using MinHash Locality-Sensitive Hashing (LSH). We set the MinHash threshold to 0.85, with 128 permutations. Second, we use MinHash LSH with the same settings to remove leaked benchmark samples based on the fixed function. Third, we prompt *gpt-4o-mini-2024-07-18* [2] to generate synthetic execution data (test cases and errors). Finally, we keep only the samples whose total token length (i.e. prompt plus answer) is shorter than 4096, measured by the model's tokenizer. Consequently, our fine-tuning dataset contains aprox. 56,000 samples [3].

Benchmarks We evaluate with two benchmarks: Defects4J [9] and GitBug-Java [17]. Defects4J comprises 835 real-world bugs from 17 open-source Java projects, from which we identify 477 single-function bugs that fit the context window. GitBug-Java comprises 199 real-world bugs, from which we identify 49 single-function bugs that fit the context window. Contrary to Defects4J, GitBug-Java is unlikely to be included in the pre-training data of CodeLLaMA since its bugs are exclusively collected from the 2023 commit history. We use the patch assessment metrics defined in RepairLLaMA [15], computing a pass@1 value for each [4].

Hyper-parameters We train a LoRA adapter with the following hyper-parameters: learning rate of 5e-4, context length of 4096, one training epoch, an effective batch size of 16, a rank of 8, alpha of 16, dropout of 0.05, and inject LoRA matrices in the *q_proj* and *v_proj* layers. Fine-tuning is conducted on a machine with 4 NVIDIA A100 40GB GPUs, and inference is performed on a single NVIDIA A100 80GB GPU. We set the inference context length to 4096 tokens, and samples 10 patches per bug with a temperature of 1.0.

5 EXPERIMENTAL RESULTS

Table 1 presents the pass@1 results for both CodeLlama-7b-Instruct and its fine-tuned version (CodeLlama-7b-Instruct-ft[4]) on two benchmarks: Defects4J and GitBug-Java. The table reports three metrics for each benchmark: plausible@1, ast-match@1, and exact-match@1.

Fine-tuning with synthetically enriched data significantly improves the performance of the model on Defects4J across all three metrics. Specifically, CodeLlama-7b-Instruct-ft achieves a plausible@1 score of 5.4%, compared with 2.4% of the baseline model. This improvement is also reflected in the ast-match@1 and exact-match@1 scores, where the fine-tuned model outperforms the baseline by 1.5 and 1.4 percentage points respectively. This result demonstrates that fine-tuning on domain-specific synthetic data can enhance the model's ability to generate quality patches.

Figure 2 shows the prompt and the exact-match patch for bug Csv-6 of Defects4J, which is exclusively fixed by the fine-tuned model. The original code failed when processing a CSV record with fewer columns than expected, resulting in an *ArrayIndexOutOfBoundsException*. The fine-tuned model successfully uses the information provided by the test case and error, fixing the issue by adding a bounds check before accessing the values array. The ability to recognize the need for this specific bounds check, addressing the issue exposed by the failing test case, likely stems from the model's exposure to similar synthetically generated execution data during fine-tuning.

Moreover, fine-tuning also improves pass@1 scores on GitBug-Java, a benchmark that includes bugs less likely to be included in the pre-training dataset of CodeLlama. The fine-tuned model achieves a 4.5% plausible@1 compared with 3.3% of the baseline model. For ast-match and exact-match the improvement is smaller, of 0.2 percentage points in each score.

[1] https://huggingface.co/meta-llama/CodeLlama-7b-Instruct-hf
[2] https://platform.openai.com/docs/models/gpt-4o-mini
[3] https://huggingface.co/datasets/ASSERT-KTH/megadiff-sf-synthetic_test_error
[4] https://huggingface.co/ASSERT-KTH/codellama-instruct-repair-synthetic-execution

Overall, our results show how fine-tuning on samples synthetically enriched with execution data enables the model to use execution information at inference time to generate correct patches.

6 RELATED WORK

6.1 Synthetic Data

Related work generates synthetic data for fine-tuning language models in general-purpose domains. Puri et al. [12] fine-tune question answering models with a corpus of synthetic questions and answers generated by GPT-2. Magicoder [20] generates synthetic instruction data to fine-tune language models for code. The Llama 3 models [6] use synthetically generated samples during post-training stages. Specifically, Llama 3 models are fine-tuned with synthetic code execution feedback, where generated code is checked for correctness by a real static analyser and unit tests, with the errors being incorporated in a self-correction loop. Our work is different for two reasons: 1) we synthetically generate execution data, 2) we focus on a different task (program repair).

Moreover, synthetic data generation is also used for pre-training language models. Namely, the phi family of models [1, 8, 10] pretrains relatively small LLM with predominantly synthetic data. Cosmopedia [3] generates millions of synthetic samples for pretraining LLMs from scratch. While such works focus on the pretraining stage, our work focuses on the post-training stage.

6.2 Synthetic Data for Program Repair

Several works have proposed self-supervised approaches for program repair, where synthetic bugs are used to train language models [2, 16, 22]. Ye et al. [23] generate synthetic bugs and incorporate real execution feedback in the samples by executing the generated bugs. Ye et al. [24] use real execution feedback as part of the loss function during fine-tuning. While existing work focus on synthetic bugs with real execution feedback, our work augments existing real bug datasets with synthetic execution feedback. To the best of our knowledge, we are the first to explore synthetic execution feedback.

7 CONCLUSION

In this work, we have shown the effectiveness of synthetic execution data in enhancing program repair capabilities of large language models. By generating high-quality synthetic test cases and error messages to augment existing bug-fixing datasets, our fine-tuned CodeLlama-7b-Instruct model achieves notable increases in pass@1 scores on both Defects4J and GitBug-Java, highlighting the model's improved ability to utilize execution information at inference time. Our results uncover the potential of synthetic data in overcoming limitations in real-world data collection for program repair tasks and offer a scalable approach to generating valuable training data.

8 ACKNOWLEDGEMENTS

The computations/data handling were enabled by the supercomputing resource Berzelius-2023-175 provided by National Supercomputer Centre at Linköping University and the Knut and Alice Wallenberg foundation. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. This work was written with help of generative AI, namely ChatGPT and Claude.

REFERENCES

[1] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219* (2024).

[2] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems* 34 (2021), 27865–27876.

[3] Loubna Ben Allal, Anton Lozhkov, and Daniel van Strien. [n. d.]. Cosmopedia: how to create large-scale synthetic data for pre-training Large Language Models — huggingface.co. https://huggingface.co/blog/cosmopedia#cosmopedia-how-to-create-large-scale-synthetic-data-for-pre-training. [Accessed 10-05-2024].

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing Chat Language Models by Scaling High-quality Instructional Conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 3029–3051.

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[7] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. 2024. Mokav: Execution-driven Differential Testing with LLMs. *arXiv preprint arXiv:2406.10375* (2024).

[8] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).

[9] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[10] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463* (2023).

[11] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A dataset of 600k java source code changes categorized by diff size. *arXiv preprint arXiv:2108.04631* (2021).

[12] Raul Puri, Ryan Spring, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. 2020. Training question answering models from synthetic data. *arXiv preprint arXiv:2002.09599* (2020).

[13] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[14] Nuno Saavedra, André Silva, and Martin Monperrus. 2024. GitBug-Actions: Building Reproducible Bug-Fix Benchmarks with GitHub Actions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 1–5.

[15] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *arXiv preprint arXiv:2312.15698* (2023).

[16] André Silva, João F Ferreira, He Ye, and Martin Monperrus. 2023. MUFIN: Improving Neural Repair Models with Back-Translation. *arXiv preprint arXiv:2304.02301* (2023).

[17] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 118–122.

[18] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Stanford alpaca: An instruction-following llama model.

[19] Shubham Toshniwal, Ivan Moshkov, Sean Narenthiran, Daria Gitman, Fei Jia, and Igor Gitman. 2024. OpenMathInstruct-1: A 1.8 Million Math Instruction Tuning Dataset. *arXiv preprint arXiv:2402.10176* (2024).

[20] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

[21] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).

[22] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*.

PMLR, 10799–10808.

[23] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[24] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*. 1506–1518.