Research Proposal for Test Transplantation

Ching-Ting Tsai
c3tsai@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Duy Phan
daniel.phan@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Yinxi Li
y3395li@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Abstract

Software testing is a crucial task in the development of software, especially for safety-critical and reliability-critical systems. Software testing is an active area of software engineering research. Software transplantation is an emerging area attracting more attention from the research community. Software transplantation techniques can be augmented to transplant unit tests from one project to another. This research project aims to automate (or semi-automate) the process of unit test transplantation, where unit tests are extracted from a donor project and transplanted into a host project. Retrieval augmented generation (RAG) techniques and available mainstream large language models (LLMs) are the backbone techniques to generate the test code that is tailored to the host project. The ideal generated test code is logically correct, executable and follows all the coding standards of the host project.

CCS Concepts

• **Software and its engineering** → **Software development techniques**; • **Theory of computation** → *Program analysis*; • **General and reference** → Empirical studies.

Keywords

Software Testing, Test Transplantation, Large Language Models, Retrieval Augmented Generation

ACM Reference Format:

Ching-Ting Tsai, Duy Phan, and Yinxi Li. 2024. Research Proposal for Test Transplantation. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software transplantation is an active research area, and when can be done, can be highly useful for developers; it will not only speed up the software development pipeline, but also release a great amount of intensive programming on the developers [1–3, 5]. Software transplantation aims to extract a feature from one system, referred to as the "donor program" and integrate it into another system, the "host program" which lacks that feature; some adaptation of the code is expected during the transplantation process [2, 4]. The techniques in software transplantation can be adapted to perform test transplantation. In this project, we will focus on building a pipeline for test transplantation, focusing on transplanting unit tests from one donor repository to another host repository. Retrieval augmented generation (RAG) and large language models (LLMs) with be the core backbone techniques to generate the test code that is tailored to the host project.

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

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Method

2.1 Core Tools

2.1.1 Database. Code data is highly dynamic and can be representative, as such, Code understanding is an important and ongoing research problem. In this research project, we will employ more complex embedding algorithms such as CodeBERT and Graph Neural Networks to capture the semantics of the code data. To facilitate retrieval, we will use ChromaDB or FAISS (or similar tools) to store unit test data. The database will store relevant information for each unit test, including the unit test function code, vector embeddings generated by a pre-trained embedding model, and metadata such as the location of the function within the repository. This allows for efficient retrieval of the most relevant unit tests based on a user-defined feature for transplantation.

2.1.2 Large Language Models and Corresponding APIs. A large language model (LLM) will play a key role in generating the transplanted unit test code. Given a feature from the donor project and relevant unit test functions, the LLM will help generate the code necessary for transplanting the unit test into the host project. We plan to use models such as Code LLaMA, GPT-4, or other state-of-the-art code-generation models to assist with the transplantation process. There are many LLMs-related tools to take a look at right now (some are somewhat stable, and many are still being developed). If time permits, we will also look at agentic AI tools such as AutoGen or Swarm (which has just been released by OpenAI).

2.1.3 Retrieval Augmented Generation (RAG).

Retrieval: Given a feature or functionality that the user wishes to transplant, we will retrieve the top k most relevant unit test functions from the donor repository using mathematical techniques such as cosine similarity to compare embeddings.

Function Mapping: After gathering the relevant unit tests from the donor repository, we will search for corresponding code functions in the host repository. These functions are mapped to the donor's relevant code to ensure a meaningful transplantation.

Augmentation and Generation: The feature code, relevant unit tests from the donor, and mapping functions in the host are then provided to the LLM.

Function Mapping: After gathering the relevant unit tests from the donor repository, we will search for corresponding code functions in the host repository. These functions are mapped to the donor's relevant code to ensure a meaningful transplantation.

Augmentation and Generation: The feature code, relevant unit tests from the donor, and mapping functions in the host are then provided to the LLM.

With carefully crafted prompts, the LLM will generate the transplanted unit test code, ensuring it is adapted to the host environment.

2.1.4 Code Graph. Code graphs are powerful tools for representing and analyzing the structure of software. In our project, we will utilize code graph tools to enhance our understanding of the codebase structure, dependencies, and relationships between different components. This will be particularly useful for:

- Identifying the relationships between unit tests and the functions they are testing
- Visualizing the structure of both donor and host projects to aid in the transplantation process
- Analyzing dependencies to ensure successful integration of transplanted tests

We plan to use tools such as Sourcetrail or CodeQL for generating and analyzing code graphs. These tools will provide valuable insights into the structure of the codebase, helping us make more informed decisions during the test transplantation process.

2.2 Data Preparation and Preprocessing

Unit tests will be extracted from open-source Python repositories (ideally actively maintained and mainstream repositories that are used by millions of developers and large companies), particularly focusing on popular frameworks for general software engineering and web development (front-end or API development frameworks). In addition, we will ensure our research will comply with the license in each repo.

Our focus will be on the test directories in each repository (which usually contains all functions used for testing and as in the pytest framework), storing all functions used for unit testing. If time permits, we might also expand to other areas such as command-line interface, machine learning frameworks, and databases; and as well, higher-level testing (file-level and system-level) will also be considered. For code parsing and code structures understanding, code graph tools and syntax trees tools will be used to traverse the code structure and extract relevant information. These extracted unit tests will serve as the primary data for our transplantation process. Extra logic will also be designed to help pinpoint the methods under test associated with the test function.

Linting and code formatting tools will be used for data cleansing (such as removing comments and making the code more consistent before adding to the vector database); in addition, the tools are also used to help make the generated tests more consistent with the host project. To enhance the Retrieval-Augmented Generation (RAG) process, we will generate natural language descriptions for each collected test, summarizing their purpose and functionality. This can be achieved using code summarization models or by extracting existing docstrings and comments. The processed code and their corresponding descriptions will then be embedded using models like CodeBERT into a dense feature space, and stored in a vector database such as ChromaDB or FAISS, facilitating efficient retrieval during the test transplantation process.

2.3 Host Program Analysis

Extra parsing logic will be designed to identify how the collected unit tests can be transplanted to potential host programs. Host programs can be ongoing high-impact projects that are actively maintained by many developers.

Some possibilities are:

- Looking at the functions and the associated contexts to identify the tests through retrieval-augmented generation.
- Scanning through the test suites and the associated contexts to identify the tests that can be transplanted.
- Applying the aformentioned methods and utilize code graphs to find the correspondence of the tests.

2.4 Prompting

Some prompting techniques will be researched to help generate the test code that is tailored to the host program. The prompt can be in various forms, in general, the prompt will contain all the necessary context for the LLM to generate the test. In addition, zero-shot, one-shot, few-shot prompting will be utilized to see how the LLMs perform when being provided with some concrete transplantation examples. The examples provided to the LLM can be the motivating examples collected from our research process, in addition, each target repo can also have its own examples. Multi-turn prompting can also be considered to refine the test generation process.

2.5 Evaluation

The test transplantation pipeline will be benchmarked to see if it can successfully transplant the tests and can outperform a regular test generation process using LLMs.

2.5.1 Manual Inspection. Manual inspection can be used to evaluate the effectiveness of the transplantation process. Manual inspections can be performed by trying to see if the code is reasonable (for example, if the basic math operations are correct or if the test is for a function that takes in a list, it should not be transplanted to a function that takes in an integer). In addition, the transplanted tests can also then be run to see if they are executable and does not cause any runtime errors in the host program.

2.5.2 Utilize Code Metrics. A combination of code metrics can be used to evaluate the effectiveness of the transplantation. Code coverage and maintainability metrics can be used to evaluate the quality of the tests. If possible, we also wants the generated tests to blend well with the code in the host program (such as following the same coding style and conventions specific to the host program).

2.5.3 Execute Transplanted Code. In a real-world scenario, making the transplanted code executable is important (also, executability can be a priority in the reranking process). The transplanted code should ideally be able to be run without any modifications from the developer. Another research path to look into here is when the generated code is not executable, we can use multi-turn prompting to refine the code to be executable.

2.5.4 LLM Utilization. Another way to evaluate the efffectiveness of the transplantation is to utilize another LLM to review the transplanted code (for example, code quality and code structure of the

transplanted code). This approach might require heavy prompt engineering to help capture all the semantics of the code structure.

References

[1] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Association for Computing Machinery, 1–10. https://doi.org/10.1109/ICST57152.2023.00008

[2] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796

[3] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. arXiv:2310.03533 [cs.SE] https://arxiv.org/abs/2310.03533

[4] Gurjot Singh Sodhi and Dhavleesh Rattan. 2021. An Insight on Software Features Supporting Software Transplantation: A Systematic Review. *Archives of Computational Methods in Engineering* 29 (2021), 275 – 312. https://api.semanticscholar.org/CorpusID:236234573

[5] Leandro O. Souza, Earl T. Barr, Justyna Petke, Eduardo S. Almeida, and Paulo Anselmo M. S. Neto. 2023. Software Product Line Engineering via Software Transplantation. arXiv:2307.10896 [cs.SE] https://arxiv.org/abs/2307.10896