Unit Test Generation using LLM and CodeQL

Group Members


```
Chinmay Saraf <csaraf2@illinois.edu>
Kedar Takwane <takwane2@illinois.edu>
```


Repository

- Code Repository:

> https://github.com/ChinuSaraf/unit-test-generation-using-llm-and-codeql

- Course Status Repository:
  - Chinmay - csaraf2
  - Kedar - takwane2

Introduction

In the ever-evolving landscape of software development, ensuring the reliability and correctness of code is paramount. One crucial aspect of this process is the **creation of effective unit tests**, which validate the functionality of individual components within a software system. Recognizing the challenges inherent in manual test creation, our project endeavors to harness the power of cutting-edge technology to automate and streamline this crucial aspect of the software development life cycle.

Project Overview

In this project, we have developed a novel tool that combines the capabilities of a Large Language Model (LLM), specifically GPT, with the insights derived from Static Code Analysis using CodeQL. The primary objective is to leverage the language model's proficiency in code generation to automate the creation of unit tests. By integrating CodeQL-generated metadata, we enhance the contextual understanding of methods and classes, enabling more informed and targeted test generation.

Motivation

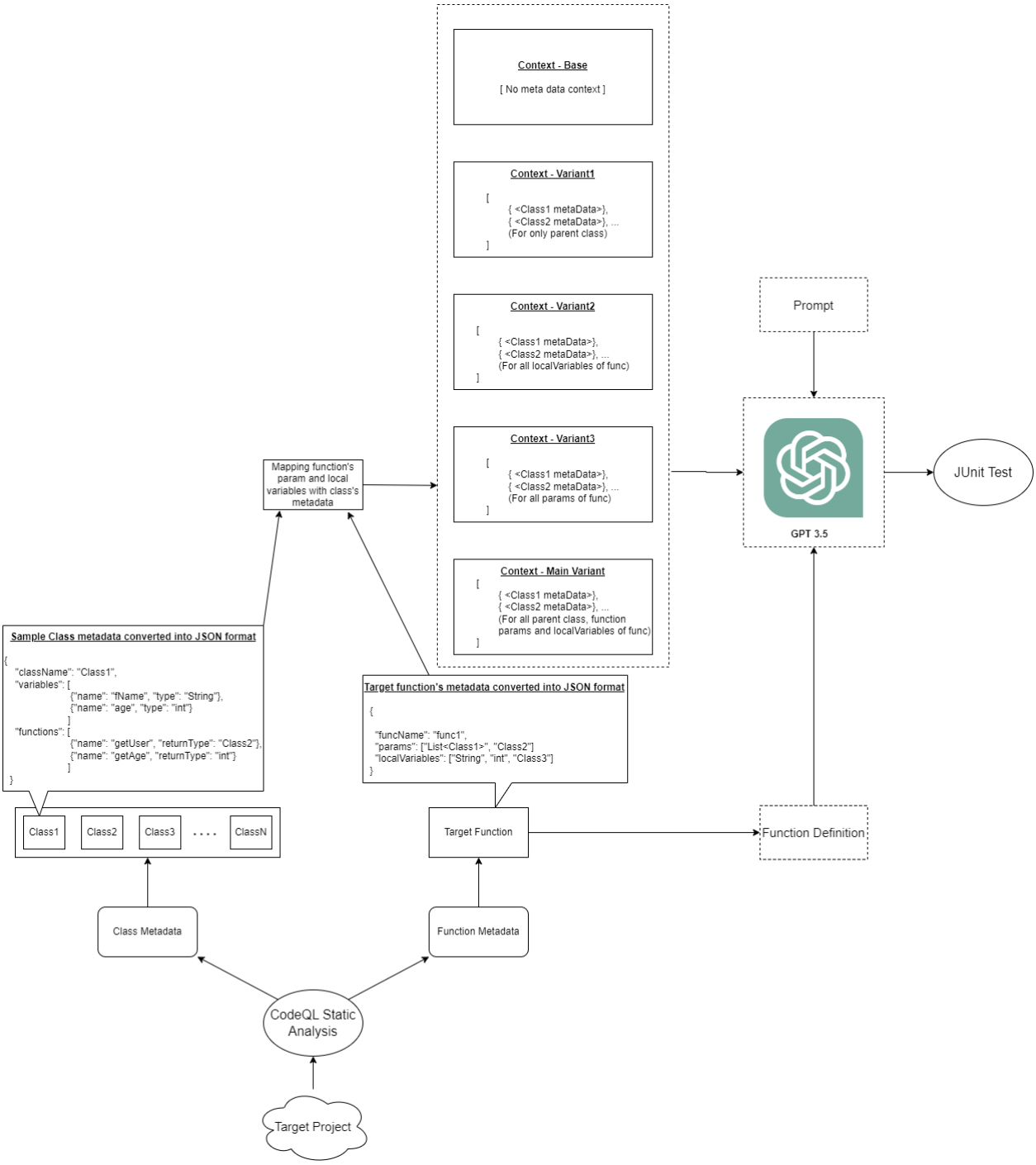
The motivation driving this initiative stems from the recognition that manual creation of unit tests can be a time-consuming and error-prone process. Developers often face challenges in understanding the intricacies of methods and classes, especially regarding mocking dependencies. To address this, we propose a synergistic approach that combines the analytical capabilities of CodeQL with the creative prowess of GPT-based LLMs. By doing so, we aim to streamline and expedite the unit testing process while ensuring the generated tests accurately reflect the intended functionality of the code.

Methodology

The tool mainly consists of two modules

1. Metadata generation using CodeQL
2. Unit Test generation using LLM model and context of metadata

The following figure shows the architecture of the tool:



1. Metadata generation using CodeQL

We have used the CodeQL static code analysis tool to generate the metadata for the project. The eventual purpose of the metadata is to pass it as a context to the LLM model to generate unit tests of the target function.

We have the following CodeQL queries:

1. get-methods-of-class.ql - This query returns all methods defined in a class and their return types.
2. get-variables-of-class.ql - This query returns all variables defined in a class and their types.
3. get-imports-of-class.ql - This query returns all imported classes and their qualified names in a class.
4. get-all-parameters-for-all-methods.ql - This query returns all parameters and their types for a function.
5. get-local-variables-for-all-methods.ql - This query returns all local variables and their types used in a function.

Sample results for all these queries for the Hadoop project are present at codeql-query-results

2. Unit Test generation using LLM model and context of metadata

1. **Parsing Metadata obtained from CodeQL:**
    - Metadata is extracted using CodeQL, exported initially as a CSV file.
    - Conversion of the CSV metadata to two JSON files:
        - `class-metadata.json`: Stores metadata linked to full qualified class names.
        - `method-metadata.json`: Stores metadata associated with full qualified method names.
    - Utilizes `convert_csv_to_json.py` script for this conversion.
2. **Unit Test Generation Process:**
    - With metadata now accessible in JSON format at `tool/output/json`, the tool proceeds to execute `run_tool.py`.
    - This script fetches metadata from the aforementioned JSON files for selected data points.

- The fetched metadata is employed as the context in different prompts provided to ChatGPT.
- ChatGPT leverages the provided context (metadata from JSON files) to generate JUnit tests.
- The generated JUnit tests are saved in `tool/output/generated-tests`.

3. **Ablation Study:**
    - The tool performs runs in various modes, conducting an Ablation study to assess the impact of different contexts on the quality of generated unit tests—more about this in the Variants section below.

This streamlined workflow showcases how the tool operates, beginning with metadata extraction using CodeQL, converting it to JSON format, utilizing it as context for ChatGPT in unit test generation, and conducting an Ablation study by running the tool in different modes.

Finalized Prompts

Here is a template of the prompt we will use in our tool: Prompt Template. The example following our prompt is the Sample Prompt Example. Here is an example of a ChatGPT Prompt.

Variants

1. No context of metadata
2. Context of metadata of parent class
3. Context of metadata of parameters
4. Context of metadata of variables
5. Context of metadata of parent class, parameters, and variables

Data

We have evaluated the tool using the Hadoop codebase. We have considered 30 data points (functions) from different microservices of the project. The data can be found at

methods.json. The data points are categorized into three types depending on the non-primitive parameters passed to the function:

1. Easy - No non-primitive parameter (10 data points - functionIds like 1, 4, 7, 10…)
2. Medium - One non-primitive parameter (10 data points - functionIds like 2, 5, 8, 11…)
3. Difficult - Two or more non-primitive parameters (10 data points - functionIds like 3, 6, 9, 12…)

Important Links

- Project used for tool evaluation -

```
https://github.com/apache/hadoop
```

- CodeQL queries - codql-scripts
- Python Script of tool - tool

Setting up the tool

CodeQL

We are using the VS Code extension of the CodeQL to work on this tool. So, ensure that you've VS Code installed in your system.

1. Setting up CodeQL CLI: CodeQL CLI Setup
2. Setting up VS Code extension: VS Code

Verify the setup as mentioned in the CLI Setup link.

Python script

Note that you need `python` and `pip3` installed before proceeding.

Install all the requirements using the `requirements.txt` file

```
pip3 install -r tool/requirements.txt
```

Setting up the OpenAI API Key

This would require an OpenAI API key which can be obtained from the OpenAI API Key. Save the key in a file named `config.json` in the `/tool` folder in the format given below.

```
{
  "OPENAI_API_KEY": "<KEY>"
}
```

How to Run?

For detailed steps, visit our GitHub repository.

CodeQL

1. Import the Hadoop database in CodeQL through the VS Code extension Hadoop GitHub Home
2. Select the **Java** language while importing the project. (This may take a while)
3. Open a command Palette (Ctrl+shift+P) in VS Code and select **CodeQL: Quick Query**. This will open the `quick-query.ql` file.
4. Copy the query from Get Class Methods and paste into the `quick-query.ql`
5. Right-click on an editor and select `CodeQL: Run Query on Selected Database`
6. This will generate an output similar to Class to Methods mapping

Run the script `run_tool.py`

```
cd tool
python run_tool.py [--mode] [--n]
```

Note: The script takes two arguments `--mode` and `--n`.

`--mode`:

- Indicates in which mode you want to run your script.
- Inputs: 0, 1, 2, 3, 4 (**Default: 4**).
- `0` -> Without Meta-data.
- `1` -> With only the method's class meta-data.
- `2` -> With only method_params.
- `3` -> With only method_vars.
- `4` -> With all Meta-data.

`--n`:

- Indicates the number of data points you want to run it on.
- Inputs: Integer between 1 and 30 (**Default: 1**)

```
cd tool
python run_tool.py
```

Results

We have used 30 data points present in this methods.json file. We have used Jacoco to evaluate the code coverage and compare different models.

Few things we considered while recording these results:

1. If assert statements are not generated by the tool, the test is marked as “not executed”, as it is the most important part of the test.
2. To make a test run, we manually made changes in a few tests including - adding missing imports, mocking some static functions/classes, etc.

Results based on the complexity of target functions

For this evaluation, we have used all 30 data points.

Easy

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
1	Passed	11	66.6
4	Passed	5	100
7	Passed	12	100
10	Passed	8	16.7
13	Passed	8	66
16	Not executed	NA	NA
19	Passed	2	50
22	Passed	5	42.8
25	Passed	4	60
28	Passed	9	53

Medium

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Passed	9	100
8	Passed	6	96
11	Passed	6	100
14	Not generated	NA	NA
17	Not executed	NA	NA
20	Passed	4	60
23	Passed	2	33
26	Not executed	NA	NA
29	Passed	0	75

Difficult

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
3	Not executed	NA	NA

6	Not executed	NA	NA
9	Passed	0	47
12	Passed	6	75
15	Not executed	NA	NA
18	Passed	17	100
21	Passed	13	100
24	Passed	7	100
27	Passed	22	37.5
30	Passed	7	100

Observations:

1. Most of the easy function tests could be executed with some changes. For medium and difficult functions, for some instances test couldn't be executed and for some, the no. of changes were significant.
2. Sometimes target functions are mocked.
3. Almost all tests had missing imports.
4. Restriction in mocking final variable types made it difficult to execute some tests. This can be improved if we provide the context of access types and static or final types of variables.
5. Due to the long context, the tool didn't generate tests for some functions.
6. Some tests were missing assert statements.

Ablation Study

For this evaluation, we have used 10 data points with functionIds - 2, 5, 8, 11…

Context with no metadata

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Not executed	NA	NA
8	Not executed	NA	NA

11	Passed	2	50
14	Passed	7	100
17	Not executed	NA	NA
20	Passed	9	60
23	Not executed	NA	NA
26	Not executed	NA	NA
29	Passed	0	100

Context with metadata of parent class

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Not executed	NA	NA
8	Passed	4	100
11	Not executed	NA	NA
14	Passed	10	100
17	Not executed	NA	NA
20	Passed	6	80
23	Passed	6	33
26	Not executed	NA	NA
29	Passed	0	100

Context with metadata of function parameter

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Not executed	NA	NA
8	Passed	4	100
11	Passed	11	50
14	Passed	10	100
17	Not executed	NA	NA

20	Passed	9	60
23	Not executed	NA	NA
26	Not executed	NA	NA
29	Passed	0	100

Context with metadata of function variables

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Passed	5	100
8	Passed	2	100
11	Passed	11	50
14	Passed	10	100
17	Not executed	NA	NA
20	Passed	1	60
23	Not executed	NA	NA
26	Not executed	NA	NA
29	Passed	2	100

Context with metadata of parent class, function parameter, and function variables

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
2	Not executed	NA	NA
5	Passed	9	100
8	Passed	6	96
11	Passed	6	100
14	Not generated	NA	NA
17	Not executed	NA	NA
20	Passed	4	60
23	Passed	2	33

26	Not executed	NA	NA
29	Passed	0	75

Observations:

1. **The correctness of the generated test has increased as we provided more and more context.**
2. **The tool managed to generate 50% more runnable unit tests when context is passed than that without context.**
3. The average number of lines of code to be changed to make the generated unit test runnable is 4.5 for variant #5 which is better than all the variants with and without context.
4. When no metadata was given, we observed that for one of the functions, the tool only generated the skeleton of the test.
5. When less context was provided, the tool faced issues in mocking the functions, variables, and class objects.

Comparison with Randoop

For this evaluation, we have used 10 data points with functionIds - 1, 4, 7,…

Our tool

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
1	Passed	11	66.6
4	Passed	5	100
7	Passed	12	100
10	Passed	8	16.7
13	Passed	8	66
16	Not executed	NA	NA
19	Passed	2	50
22	Passed	5	42.8
25	Passed	4	60

28	Passed	9	53

Randoop

Function Id	Test Status	No. of lines changed	Line Coverage (in %)
1	Passed	0	0
4	Passed	0	0
7	Passed	0	67%
10	Passed	0	67%
13	Passed	0	0
16	Passed	0	57%
19	Passed	0	75%
22	Passed	0	0
25	Passed	0	0
28	Passed	0	0

Observation:

- **Tests generated by Randoop cover some of the 10 methods under consideration.**
- For some, it fails to generate tests that can cover the methods under test and it generates tests for other functions/constructors from the target class.
- We tried to pass the methods under test as a `methodlist` argument to the randoop command, yet the tests generated did not cover them.
- In comparison, our tool generated tests for most of the functions, and after a few changes, we were able to run the test and get more than 50% coverage for the methods under test.

Notable Observations

1. The correctness of the generated test has increased as we provided more and more context.

2. The tool managed to generate 50% more runnable unit tests when context is passed than that without context.
3. The metadata of the parent class, function parameter, and local variables has improved the generation of the unit tests.
4. Functions with less non-primitive parameters have more accurate generated unit tests.
5. Metadata about the access type of the function can be helpful in unit test generation. This could be a good future scope for this tool.
6. Metadata of whether a method is static or not can be useful.
7. Giving context only specific to the functions and variables used by the target function can improve the tool.
8. We can also pass the value of the constant and final variables to improve the test generation.
9. For Randoop test generation we observed for certain cases it can generate comprehensive tests that can cover and test the target method (see example `19`) and sometimes it fails to generate tests that cover the target methods (see example `25`).

Conclusion

In conclusion, our project introduces a groundbreaking approach to address the challenges of manual unit test creation by synergizing a Large Language Model (LLM), specifically GPT, with insights from Static Code Analysis using CodeQL. The integration of these technologies has proven effective, as evidenced by the increased correctness of generated tests with added context, the enhanced metadata from CodeQL contributing to targeted test generation, and the tool's superiority over Randoop in test coverage. The observations highlight the potential for future improvements, such as incorporating access type and method staticity metadata, focusing context on the target function, and passing constant and final variable values for further test refinement. Overall, our project provides a significant step forward in automating and expediting the unit testing process, offering a promising solution to the time-consuming and error-prone nature of manual test creation in software development.