Commit-aware Selective Mutation Testing

Kyungwook Nam
KAIST
ruddnrdl@kaist.ac.kr

Juchang Lee
KAIST
c2w2m2@kaist.ac.kr

Min Woo Baek
KAIST
qqqor@kaist.ac.kr

Hyeongjin Choi
KAIST
gagu19x99@kaist.ac.kr

Abstract

Mutation testing is one of the most powerful testing methods that can check the quality of the existing test sets. However, In today's frequent distribution situations using CI/CD, naive random mutation testing is neither feasible nor scalable as it creates mutations not only from changed code but also from the parts irrelevant with the changes. As a result, many studies related to the reduced cost of mutation testing are underway, but we can't use them directly in CI/CD because most of them depend on machine learning or the whole execution of mutation. We start with changed parts of the source code and gradually expand to parts that are relevant to changes. By only creating mutations from the "commit-relevant" part, our method can reduce the cost of mutation testing and we think this is more suitable testing for the CI/CD environment.

1 Introduction

For determining the quality of test sets, mutation testing is often used. The technique intentionally creates a mutant by modifying a small part of the original code and compares the result of the two. However, as most situations nowadays go under CI/CD, applying naive random mutation testing is not an attractive choice. As this creates mutations not only from changed code but also from the parts irrelevant to the changes. It is a waste of resources to check the already checked part of the source code and does not offer any information about changes. As a result, prior works tend to reduce the cost of mutation testing. These works mainly follow the two strategies below.

1. **Mutation Reduction**: Reducing the number of mutants to test without losing the efficiency of the test. Mutant sampling, mutant clustering, and high-order mutation would be examples.

2. **Execution Cost Reduction**: Reducing the execution cost of testing a single mutant. Strong, weak, and firm mutation, run-time optimization, and advanced platform support would be examples.

In this project, we reduce the cost of generating mutants by applying mutation only to parts that are actually relevant to the commit. We call this *commit-aware*, and the definition of this is referred to as "a mutation-based assessment metric capable of measuring the extent to which the program behaviors affected by some committed changes have been tested", according to [1]. There have been several studies related to conducting mutation testing based on commits, such as [1] that proves there are mutants that can be stated to be more relevant to the commit than others, and [2] that applies machine learning to define which mutant is commit-relevant. However, it is difficult to apply them in CI/CD environments as they either require heavy resource consumption or all the generated mutants have to be executed first to be evaluated. Therefore we propose a static approach that analyzes the code base which can identify parts that are commit-relevant and generate mutants by applying mutation operators only to these parts. We start with the committed code of the source code and gradually expand to parts that are relevant to the changes. To check whether our method works, we also conduct naive random mutation testing and compare it with our method. We also check the rate of commit-relevance [1] of created mutations. We open-source our prototype at https://github.com/nkwook/commit-relevant-mutation-testing.

2 Related Work

We can see the definition and effectiveness of commit-relevant mutation testing from a previous study [1]. However, in that paper, we only can check if a given mutation is commit-relevant only after executing that mutation and comparing it with the result of the original code. It is necessary to select a commit-relevant mutation without executing it. In this regard, the paper of [2] specifies key features related to commit-relevant and presents a method of determining whether the mutation is commit-relevant or not, using machine learning without executing tests.

[2] introduces some dependency graph-related features to get contextual information from the changed code. These dependency graphs are as follows: data dependency (direct data dependency, indirect data dependency), control dependency, and control flow. Referring to those features, we thought of a way to find parts related to the changed code without utilizing machine learning. We also measured precision and recall of the commit-relevant [1] mutations to see if the mutations

generated by our method are really effective.

3 Background

3.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) serves as a fundamental concept in the field of computer science, specifically in programming language theory and compilers. It represents the hierarchical structure and semantics of the source code, serving as an intermediate representation that aids in program analysis and transformation. The AST is a data structure that represents the syntactic and semantic structure of source code, allowing for efficient manipulation and analysis by compilers, interpreters, and various program analysis tools. The construction of an AST involves parsing the source code and transforming it into a tree-like structure that captures the program's hierarchical organization. Unlike the concrete syntax, which includes all the details and intricacies of the programming language's grammar, the AST abstracts away unnecessary information, focusing solely on the program's structural elements and their relationships. The AST consists of nodes representing different language constructs such as statements, expressions, functions, and declarations, with edges denoting the relationships between them. Each node in the tree corresponds to a specific syntactic construct in the source code, and the tree's structure mirrors the nesting and scope of these constructs. By traversing the AST, it is possible to perform a wide range of analyses, optimizations, and transformations on the program.

3.2 Use-Define Chain

A Use-Define Chain (UD chain) is a fundamental concept in program analysis that helps in understanding the flow of data within a program. It establishes a relationship between the use of a variable or an expression and its definition or assignment in the program. By tracking this relationship, program analysts can infer valuable information about the values assigned to variables and their impact on subsequent computations.

The UD chain consists of two essential components: a use and a define. The use refers to a point in the program where a variable or expression is being referenced or read. It represents the consumption of a value. On the other hand, the define corresponds to a point in the program where a variable is assigned or defined. It represents the production or creation of a value. By linking the use of a variable to its corresponding define, the UD chain captures the flow of data, allowing analysts to reason about the possible values that a variable can take at different points in the program.

The UD chain serves as a powerful tool for various program analyses. For instance, in data flow analysis, it helps identify variables that are used before being defined, indicating potential data dependencies or uninitialized variables. Reaching definitions analysis employs UD chains to determine the set of definitions that can reach a particular program point, enabling the identification of redundant assignments or dead code. Furthermore, optimization techniques such as common subexpression elimination and constant propagation heavily rely on UD chains to identify opportunities for code simplification and performance enhancement.

4 Design

This section describes how we designed our method of applying mutation operators only to codes that are relevant to the commit. Figure 1 shows an overall overview of how our system works. The system takes a commit with a designated parent and child as input and continues to process it as an AST. Based on the dependencies of variables used in the commit, the system further marks relevant nodes. The scope of our work is limited to Python, therefore we utilize the provided AST module to construct and apply analysis on top of it.

4.1 Dependency

Initial Analysis on Commit: To find codes that are relevant to the commit, we first analyze the committed code itself. We extract contextual information such as what variables were modified and the scope of the commit applied.

Data Dependency: With the AST obtained, we traverse each function definition in a linear sweeping manner to identify all UD chains in it. We locate the definition or assignment and the usage of variables to do so. The traverse unit is in functions as variable names can overlap among different functions although they do not share dependencies. During the traversal, data dependencies are extracted by examining the relationships between variable uses and assignments. When encountering a variable use node, the corresponding variable name and its defining assignment nodes are recorded. This information establishes the data dependency relationship, illustrating which assignments are associated with each variable use. We gather the collected data dependencies and tag them by the scope (function in this context). The dependencies are later used to mark nodes within the AST based on what variables are collected in the previous step.

Control Dependency: We collect control dependency in a way similar to how we collected data dependency. Again traversing each function definition, we observe nodes that have the 'test' attribute indicating a branch condition. Control dependency can be defined in many ways. However, in our work, we say that there exists a control dependency between two variables if one is used in the condition statement and another is used within the body of the condition branch.

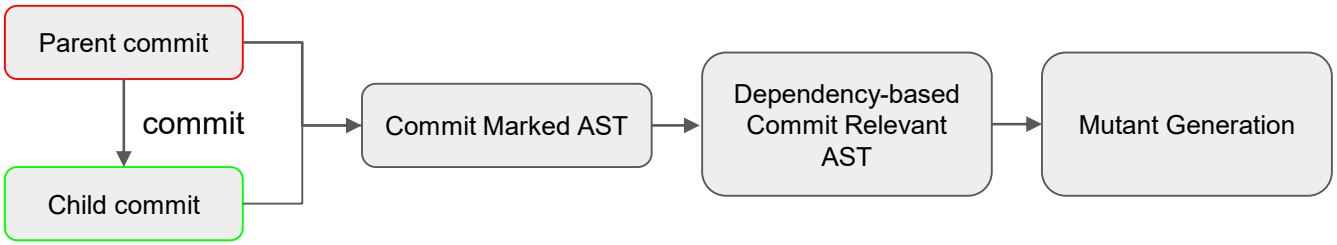

Figure 1: Overview of system

4.2 Dependency Applying Algorithms

We propose 4 algorithms for marking nodes using data and control dependency.

- **A1:** Commit Only
- **A2:** Commit + DD
- **A3:** Commit + DD + CD
- **A4:** Commit + DD + CD+

The "Commit" indicates the committed code while "DD" and "CD(+)" indicate the code parts that have data dependency and control dependency respectively. We implement two methods of applying control dependency to the code; CD and CD+. The difference is as the following.

- **CD:** Marking only the test statement.
- **CD+:** Marking statements that include the usage of variables within the test statement.

We believe these two methods have their own pros and cons. As the granularity of deciding which node is relevant to the commit is quite coarse, we tried changing the size of the scope that we search for commit-relevant code. How the results differ by the methods will be described in the following section.

5 Implementation

We implemented our system with 2K LoC of Python. To create the AST of Python code, we used the ast module. As mentioned in Section 1, please refer to our github repository for implementation details such as mutation testing and diff processing.

5.1 Mutation Operators

The types of mutation operators we used for mutation are as follows.

1. Conditionals Boundary Mutator: Replaces relational operators as the following.
   - < → <=
   - <= → <
   - > → >=
   - >= → >
2. Increments Mutator: Replaces assignment increments with assignment decrements and vice versa.
3. Invert Negatives Mutator: Inverts unary negations.
4. Math Mutator: Replaces binary arithmetic operators as the following:
   - + → -
   - - → +
   - * → /
   - / → *
   - % → *
   - & → |
   - | → &
   - ^ → &
   - « → »
   - » → «
5. Negate Conditionals Mutator: Replaces all conditionals as the following:
   - == → !=
   - != → ==
   - <= → >
   - >= → <
   - < → >=
   - > → <=
6. False returns Mutator: Replaces return values with False.

Table 1: Description of test suite used in experimental setup. All are built from scratch.

Testcase	Explanation	Change with commit	# Test	Prev LoC	Post LoC
Case 1	Multiple if conditionals	add if statement with another condition	10	18	26
Case 2	Fibonacci Sequence	edit if statement	7	13	17
Case 3	Bubble, Merge, Quick Sort Algorithm	remove if statement	7	52	49
Case 4	Arithmetic + Conditional statements	add if statement and variable assignment	10	37	44
Case 5	Arithmetic + Function call	add arithmetic and if statement	15	29	36
Case 6	Arithmetic + Function call	add arithmetic and change if statement	7	32	39
Case 7	Array indexing + Arithmetic statements	edit constant variable value	6	15	15

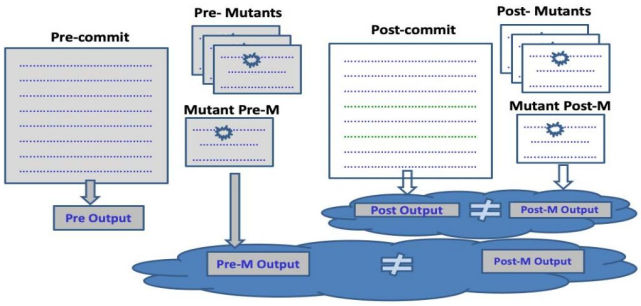

Figure 2: Concept of commit-relevant mutants. A mutant is relevant if it impacts the behavior of the committed code and the committed code impacts the behavior of the mutant. [1]

7. True returns Mutator: Replaces return values with True.
8. Null returns Mutator: Replaces return values with None.
9. Bitwise Operator Mutator: Consists of three sub-mutators that respectively reverse bitwise operators, replace a bitwise operation by its first member, and by its second member.
10. Constant Replacement Mutator: Mutates inline constant. The mutator is composed of 6 sub-mutators that mutate constants in different ways; a given constant c is mutated to c, 1, 0, -1, -c, c$+1$, c-1 respectively by each mutator.

5.2 Testcases

We used a total of 7 test cases to test our system in different situations. We describe the test cases in Table 1.

6 Evaluation

6.1 Research Objectives

We aim to answer the following two research questions.

- **RQ1: Can our system find commit-relevant mutants?**
- **RQ2: How effective is it using the commit-relevant mutants?**

For RQ1, we compare the results of our system with those from manually created commit-relevant mutants. We evaluate the recall and precision based on the manually confirmed ground truth.

For RQ2, we evaluate the efficiency of mutation testing when using the commit-relevant mutants created from our system. We compare our system with random mutation testing in the aspects of the mutation score, and number of mutants used.

6.2 RQ1: Existence of commit-relevant mutants

For ground truth, we follow the approach from [1] and compare the test result of pre-commit mutated code, post-commit mutated code, and post-commit original code. As Figure 2, mutants are commit-relevant when the output of pre-commit mutated code and post-commit mutated code are different, and the output of post-commit mutated code and post-commit original code are different as well. After the mutation test execution, we gain a kill matrix that contains the result of whether each test for a specific mutant has been killed. Then by comparing the results of the pre-commit and post-commit versions, we can obtain information on commit-relevant mutants. Of course, directly modified parts are excluded from the analysis. For checking the ability of our algorithms at detecting commit-relevant mutants, we apply algorithms S3 and S4. After gaining commit-relevant mutants using each algorithm, we calculate the precision and recall.

Table 2 shows the result of the existence experiment of commit-relevant mutants. The columns "P" and "R" represent precision and recall respectively. Genuine commit-relevant mutants are observable in four out of seven code samples. Three code samples do not contain commit-relevant mutants, which means changes in corresponding code samples are not sufficient to affect mutation test results. This is possible since we do not have a representative test suite for benchmarking, and built ours from scratch.

If our algorithm is effective to select commit-relevant mutants, we expect that algorithms result in fairly high recall and higher precision than entire testing. However, from precision and recall values evaluated from the other four sample codes, we demonstrate that our algorithms do not outper-

Table 2: Investigation on the ability to mark commit-relevant mutants by using S3 and S4. P denotes precision and R is for recall in the table.

Sample	Ground Truth				S4			S3		
	# Mutants	# Relevant	P	R	# Mutants	P	R	# Mutants	P	R
Conditionals1	80	8	0.1	1.0	50	0.1	1.0	35	2.86e-2	0.125
Fibonacci	6	0	0	0	6	0	-	6	0	-
Sort	30	15	0.5	1.0	22	0.409	0.6	9	0	0
Simple	79	0	0	0	64	0	-	43	0	-
Conditionals2	95	0	0	0	94	0	-	19	0	-
FunctionCall	74	29	0.391	1.0	69	0.391	0.931	38	0.184	0.194
List	36	25	0.694	1.0	32	0.656	0.84	22	0.545	0.48

Table 3: Evaluation results of the four algorithms proposed in this work. # Mutants denotes the number of mutants and MS stands for mutation score.

Sample	Non Aware		S1	S2		S3		S4	
	# Mutants	MS	# Mutants	# Mutants	MS	# Mutants	MS	# Mutants	MS
Conditionals1	80	57.50%	6	6	66.67%	35	54.29%	80	100.00%
Fibonacci	6	66.67%	6	6	66.67%	6	66.67%	6	66.67%
Sort	30	86.67%	6	6	100.00%	9	88.89%	22	86.36%
Simple	79	88.61%	**22**	**38**	78.95%	43	81.40%	64	85.94%
Conditionals2	95	63.16%	10	10	40.00%	19	52.63%	94	62.77%
FunctionCall	74	50.00%	23	27	29.63%	38	34.21%	69	52.17%
List	36	61.11%	5	5	**00.00%**	22	**54.55%**	32	68.75%

form entire mutation testing. In the case of A3, recall is too low to find enough amount of commit-relevant mutants. A4 showed higher recall but since CD+ method which marks data-dependent nodes of control-dependent marked ones covers almost every node, it lead to nearly the same level of precision for the entire mutation testing.

Here are some reasons that our algorithm couldn't achieve a sufficient level of selectivity. One notable thing is the portion of commit-relevant mutants in our sample test suite was relatively higher than [1]'s. We guess the difference was oriented since we use way more simple test suites and sample codes in this work, leading to a single mutation giving a bigger effect. Also since the test suite was too simple, all the CD+ methods propagate to the entire code. Due to the coarse-grained scope of our implementation, we are not yet able to distinguish nested conditionals from sequentially aligned ones. Also, more test cases are needed to have statistical effectiveness.

6.3 RQ2: Analysis of efficiency and selection tendency

While we could not achieve dominantly effective commit-aware testing results, analyzing the behavior of four algorithms and the difference between them will help to understand the nature of commit-relevant mutation testing.

Results in Table 3 provide the number of mutants that the algorithms proposed cover, and the mutation score of each method. Despite it does not directly detect the commit-relevant mutants from [1], particular methods capture the data dependency and control dependency between AST nodes and make it able to test commit-relevant mutants which are not included in direct modifications.

Case Study: The "Simple" sample and the "List" sample are notable examples that demonstrate that different dependencies capture different features of the code. When comparing S1 and S2, apart from the "Simple" sample, the number of mutants merely changes. Then considering S2 and S3, the "List" sample shows the most drastic change in the number of mutants created.

In addition, we can see that the more the number of dependencies is considered, the more the number of mutants is generated. This presents the trade-off between efficiency and cost.

7 Discussion

7.1 Limitation

The system's ability to find commit-relevant mutants is limited and the algorithms used in the system do not surpass the performance of entire mutation testing. However, the analysis of different dependencies and their impact on mutant generation provides insights into the behavior of commit-relevant mutation testing. The findings emphasize the trade-off between efficiency and cost in considering various dependencies. Overall, further improvements in the system's selectivity and the inclusion of more comprehensive test cases are necessary for more effective commit-aware testing.

7.2 Future Work

For future work, we aim to focus on three aspects. First, the reliability of the tests and evaluation methods. Since there was no precedent test suite for our novel static commit-relevant testing approach, the code base we constructed has some discrepancies between diverse code bases in the real world of software engineering. We'll try to build the test suite in a concrete manner that can validate the properties of each dependency marking without ambiguities. Also, since we haven't tested many cases, our algorithm has vulnerabilities that affect the mutant selectivity. For instance, the S4 algorithm cannot distinguish nested conditionals from sequentially aligned ones. We'll improve the confidence level of our algorithms through more test cases.

Next, for the system to support commit-relevant mutation testing on real-world code bases, more feature coverage is inevitable. For example, cases like commits on multiple files, and more mutation operators.

Lastly, the ultimate goal would be the integration to the existing CI/CD pipeline such as GitHub actions, and make a meaningful impact on reducing mutation testing costs and maintaining the test reliability.

8 Conclusion

In this project, we propose a method to generate commit-relevant mutants by analyzing the source code in a static approach. We collect data dependency and control dependency of the code and apply them to mark the AST in 4 different algorithms. As a result, we demonstrated that creating commit-relevant mutants with static analysis is feasible and the mutation score tested with it remains considerable.

9 Acknowledge

This is a CS453 project work.

References

[1] Wei Ma, Thomas Laurent, Miloš Ojdanić, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–405, 2020.

[2] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. Mudelta: Delta-oriented mutation testing at commit time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 897–909, 2021.