Using Co-evolution of Artefacts in Git repository to Establish Test-to-Code Traceability Links on method-level

Research paper

Yichao Xu

ABSTRACT

Test-to-code traceability links describe the test relationship between the test codes and the product codes. It is beneficial for maintaining the test codes during developing, but to manually create and manage such links are time-consuming and fragile. The automatic link establishing strategies can properly solve such an issue.

Our works analyse the feasibility of using the co-evolution relationship in the code repositories to establish links between test and tested codes. Because the performance of such an idea depends on how the developers maintain their test codes, we mined six projects to observe how their developers did. The results demonstrated the rareness of such a relationship, which implies the bad performances of the idea.

We still implemented the idea, Co-Evolution. Besides, We tried three optimisations with the enlightenment in the previous research: co-creation relations, the commits filters, and the APRIORI algorithm. We applied all of them to three large, well-studied open-source Java projects. The precision, recall and f1 score were used as performance measures during the experiments. The results were bad as expected for Co-Evolution and it shown a significant boost after filtering. However, they were still low compared with classic ideas.

1 INTRODUCTION

Test-to-code traceability link is a way to model the relationship between the test codes (i.e. tests) and the product codes (i.e. functions). This kinds of links are helpful for developing, maintaining, and testing. As for the developing, a unit test is a continuously updating document of a certain function for its common user stories [1]. The test-to-code traceability links imply them test relationships between functions and tests, which make the codes more understandable. As to the maintaining, the unit tests may need to be refactored or even be re-engined after changing the tested functions [1], so the possible issues will occur if the developers unexpectedly forget to change some of them. Well-maintained traceability links can help to avoid this kind of problems because these links precisely indicate the tests of each function. Regarding testing, these links provide the developers with a more specific view about the validation of the software [2], compared with the line or the branch coverage.

It is time-consuming to maintain such links manually during development. Hence, a better way is to automatically establish them when needed. There has already been some widely-used technique. The most common ones are String-Based approaches like the "Naming Convention" (i.e. NC). In NC, the tests and functions are linked if their names following a simple pattern. For example, the function "A" and test "testA" will be linked. There still are Invocation-Based approaches. One example is "Last Call Before Assert" (i.e. LCBA), which will establish links if a function is invoked at the end of a test. Except for the two, many Statistical Call-Based approaches were also proposed in previous researches. All of they base on own assumption to the developers of a project. Specifically, the String-Based ones rely on that the developers named the tests and functions following specific patterns. The Invocation-Based and Statistical ones assume the particular invoking locations or frequency of functions in the tests. Because not all developers follow these assumptions, their performances fluctuated in different projects. Even if the current approaches have shown high performance in many projects, it is still valuable to investigate the possible new approaches. Because they may be more suitable in some cases. The idea of the CoEv focuses on the changes in the project's repository instead of the codes. It is potential to find out the hidden links from the co-evolution relationship. Moreover, White [3] implemented the TCtracer, which uses different approaches to score links and then choose the links with high scores. The idea addresses the issues from the unexpected programming customs by mixture results of different approaches. The tools like TCtracer demand more approaches based on various assumptions. Therefore, our research on the CoEv is valuable to it.

As to our works, we implement the CoEv on the method-level and three optimised approaches. Our research questions are about the feasibility of the co-evolution-based idea and the performances of the implemented approaches. To investigate the usability, we analyse how the developers maintained the tests in six open-source Java projects. For measuring the performances, these approaches are applied to the repositories of three widely-used open-source projects. The precision, recall and f1-score are used to evaluate their performance. The main contributions of the paper are:

- An analysis of the test maintenance for six open-source Java projects.
- An tool for establishing test-to-code traceability links between the test methods and tested functions by the CoEv approach.
- An evaluation for the performances of these strategies on the method-level to three open-source Java projects.

The remained parts of the paper is based on the structure below: Section 2 discusses the previous the related research about the co-evaluations; Section 3 demonstrates all techniques used and the implementation of them in the tool; Section 4 discusses the experimental design, the experimental implementation and the findings from the experiments; Section 5 compares the co-evaluation methods with other strategies; And Section 6 concludes the whole paper.

2 BACKGROUND

Regarding the previous works, the most straightforward idea to establish a test-to-code traceability link is to manually identify the links after creating tests and also manually update them after

changing. The idea was actually suggested by many standards and papers [2, 4, 5], but the manually-maintained links are fragile, error-prone and time-consuming [2, 3, 6, 7]. Various automatic approaches have already been proposed for solving these issues. The most classic ones are "Naming Conventions" (NC) and "Last Call Before Assert" (LCBA): NC assumes that the name of a tested function is similar to the names of its tests, like the 'functionA()' and the 'testFunctionA()'; LCBA believes that a tested function is always invoked at the end of its tests. In addition to the two common approaches, there are "Statistical Call-based Techniques" (SCTs) and the "Co-Evolution" (CoEv) approaches. SCTs uses the statistical data to calculate scores for the possible links and then filter invalid ones out by a predefined threshold. White [3] has implemented such kind of approaches based on the Tarantula fault localisation and Term-Frequency-Inverse Document Frequency (TFIDF). CoEv mines the repository of the target software to find out all functions and tests changed in the same commits, and links all of them. It was implemented by Rompaey on the class-level[7], but the approach performed with worse precision and recall than others.

In terms of the related works, our approaches are enlightened by the research approaches and the findings from the papers about the co-evolution of the software artefacts. As to their approach, both Marsavina [8] and Vidacs [9] used the APRIORI algorithm to find out the association rules for the changes occurred on linked functions and tests. We "reverse" such an idea and apply the algorithm directly to all tests and functions in a project. After that, the association rules between tests and functions are identified as the traceability links. Besides, Zaidman[10] analyse the test behaviours in three open-sources projects by the co-evolution of test and product code. We partly redo their experiments by using a visualised technique, "Change History View", to analyse how the developers in six open-sources Java projects maintained their test codes. As for the result, Marsavina [8] released a series of co-evolution patterns between test codes and product codes. We use the co-creation pattern to establish the traceability links. Such a pattern implies that a tested function is frequently co-created with its tests in most projects.

3 APPROACH

Our works focus on the CoEv approach on tests-level. We divided it into two steps: The first one is to mine the code repository to obtain the changes in each commit; The second one is to establish the links from the co-evolution of tests and functions. In the following parts of the section, we discuss the two steps respectively.

3.1 Repository Mining

Figure 1 shows a process for repository mining. We extract the changes on each commit from the Git repository, and then stores them into a database. Three different programs are used:

PyDriller. [1] It is used to mine the changes on each commit of a repository, including changed files, tests and functions. However, the tool cannot identify the specific types of these change, which makes impossible to track them after renaming.

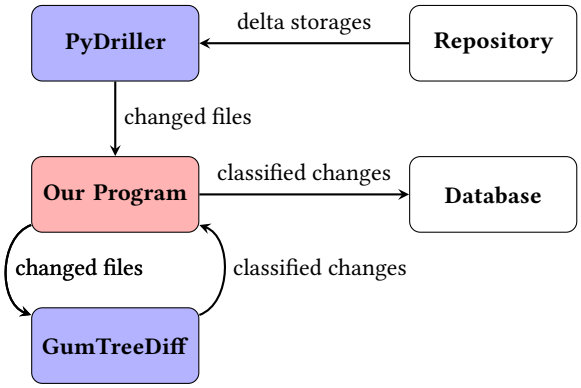

Figure 1: Processes for changes mining

GumTree. [2] It is used for identifying the type of changes. The tool is similar to the DIFF, but it can provide us with more information such as adding new local variables, removing a test invocations, or modifying the value of a certain field [11]. In our approach, we classify changes into the following four categories:

- *ADD*: A new test or a new function is created.
- *MODIFY*: The codes of a test or a function are changed.
- *RENAME*: The signature of a test or a function is changed.
- *OTHERS*: The changes to global variables, the class variables, or the interface etc.

Our Program. It handles the interaction between the *PyDriller* and the *GumTreeDiff*. After *PyDriller* finds out the changed files in a commit, these files will be sent to our program, and then the *GumTreeDiff* will be invoked to extract the fine-grained changes from these files. Then, our program classifies all of these changes and puts results into a database.

3.2 Link Establishing

"Co-Ev" approach links tests and functions changed in single commit, which relies on the intuition that the tests and the functions are evolved together. Because of its bad recall and precision value comparing with others, we also implement the following three optimised approaches.

Co-create pattern. As we discussed in previous sections, Marsavina [8] and Vidacs [9] find out a co-creation pattern which implies a strong relationship between the creating of a function and its tests. According to the pattern, it is reasonable to deduct that a function is typically created with its tests in the same commit. Therefore, we link all tests and functions, which are created in single commits.

Commits filters. During the researching process, we find that hundreds of or even thousands of unrelated tests and functions are changed in some commits. It is a common situation during developing. For example, the developer added the keyword final to nearly all tests at a single commit in the repository of the Commons-Lang. Understandably, such kind of commits can lead to a broad set of mistaken links, so we applied a set of filters to find out these abnormal commits and ignore them during link establishing.

[1] PyDriller: https://github.com/ishepard/pydriller

[2] GumTreeDiff: https://github.com/GumTreeDiff/gumtree

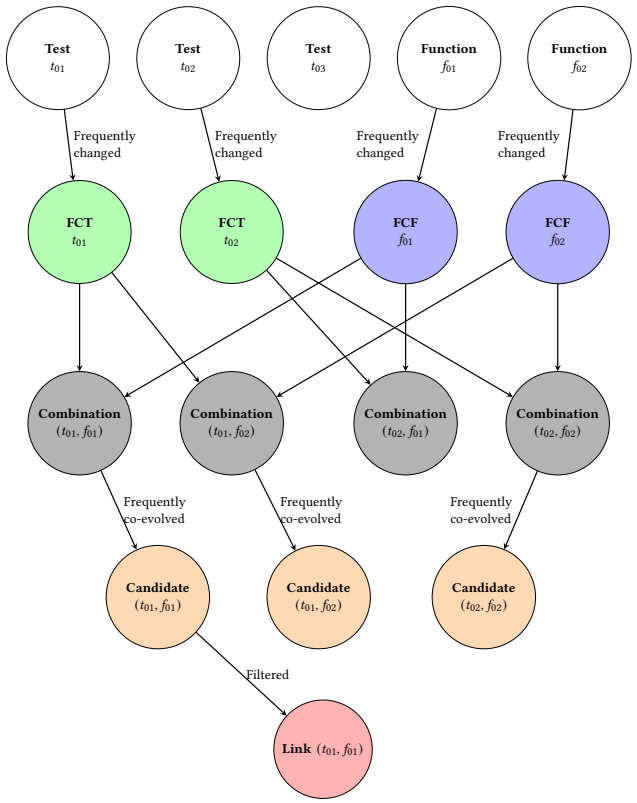

Figure 2: Illustration for APRIORI Algorithm

APRIORI Algorithm. The last one comes from the testology in [8] and [9]. The original version of the algorithm is used to mine the association rules for items from a large number of transactions [12]. In the case that two entities are associated, it is reasonable to believe that they will occur again in future transactions. In our optimisation, the changes to codes and commits are seen as the items and the transactions, respectively. We apply the algorithm to find out the association rules between the changes to both of the tests and the functions. Such a kind of rules implies a strong co-evolution relationship between the tests and the functions. According to our previous assumption, we believe that the test and the function in a traceability link are frequently changed, so it is reasonable to link tests and functions which are associated together.

Figure 2 illustrates how the algorithm predicts a link. Specifically, there are four different steps: Firstly, the algorithm finds out all frequently changed tests (FCT) and frequent changed functions (FCF). Two thresholds, "*min test changes support*" and "*min function changes support*" are used in here to define the FCT and FCF. A test (or a function) will be classed as frequently-changed ones, if the number of commits where it was changed is larger than the "*min test changes support*" (or the "*min function changes support*"). In Figure 2, there are three tests and two of them are FCTs (t_{01} and t_{02}). There are two functions and all of them are FCFs (f_{01} and f_{02}). Secondly, the algorithm constructs a series of combinations by joining each FCT and FCF together, which is similar to the Cartesian product for a set of FCTs and a set of FCFs. In Figure 2, there are four combinations, (t_{01}, f_{01}), (t_{01}, f_{02}), (t_{02}, f_{01}), (t_{02}, f_{02}), because of the two FCFs and the two FCTs. Thirdly, the algorithm finds out all combinations in which both the FCF and FCT are frequently changed. These combinations are seen as candidates for the links. We use a threshold, "*min coevolution support*", to identify whether a combination is a candidate or not. If it is, the number of the commits, where both FCF and FCT in the combination were changed, should be larger than the threshold. Finally, the algorithm will output any candidate link whose confidence value is greater than the "*min confidence*". The number is a threshold to limit the confidence value of the predicted links, and we use the following equation to calculate the confidence of a candidate link.

Threshold	Value
$min\_function\_change\_support$	$Median_{functions}$
$min\_test\_change\_support$	$Median_{tests}$
$min\_coevolution\_support$	$Median_{coevolution}$
$min\_confidence$	0.5

Table 1: Values for all thresholds

$$Confidence = \frac{\#OfCoEvolvedCommits}{\#OfFunctionEvolvedCommits}$$

The numerator is the number of commits where both the FCF and FCT in the candidate link were changed. And the denominator is the number of the commits in which the FCF was changed. In Figure 2, there is only one remained candidate which satisfies the limit for min confidence, so the algorithm will only outputs the (t_{01}, f_{01}) as a predicted link. The algorithm uses total four thresholds, and Table 1 shows the values of these thresholds that we used during the experiments. Specifically, as for three *support_number*s, the median are used, because it filter out the lower half from the dataset. As to the confidence, we simply use the 0.5 as universal threshold for all projects.

4 EVALUATION

In this section, we discusses the research questions, experiments design, and their result.

4.1 Research Questions

Our first research question focuses on how developers maintained the tests. As we mentioned, CoEv is based on the assumption for co-evolution, which is directly affected by the developers' behaviours when they maintain the tests. Specifically, suppose they always change the product codes and the test codes in different phases. In that case, the approach will undoubtedly show a bad performance. It is reasonable to estimate that the approach will perform with high precision and recall if the developer continually updates or creates the tests with the functions together. The second one is about the performance of all approaches that we implemented.

In conclusion, the two research questions are list as below:

- **RQ01**: How the developers typically maintain the tests in the real-world projects ? Is the tests co-evolve with the functions frequently?

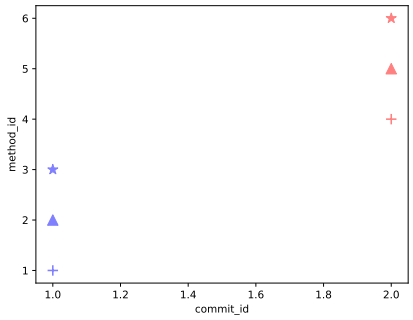

Figure 3: Example for Change History View

- **RQ02**: What are the performance of the co-evolution, co-creation, commits-filtered co-evolution and APRIORI approach in method-level?

4.2 Experiments Design

Tests Maintenance Behaviour Analysing. For investigating the research question, we first counted the number of different commits. These figures numerically demonstrates the frequency of the co-evolution. And then, we used the "*Change History View*". It was used by Zaidman[10] for analysing developers' behaviours when they maintain test codes. Such a diagram provides us with a visualised and general view about the changes of tests and functions based on commits. Figure 3 shows the example for such a view. Its x-axis is the id for each commit, and y-axis is the id for each method. The shape of point means different types of changes: + is *ADD-METHOD*, Δ is *MODIFY-METHOD* and ⋆ is *RENAME-METHOD*. Besides, the different color can be used to identify the changes to the tests and the tested function: red points are for the functions, and the blue ones are for the methods.

Approaches Performance Measuring. In the experiments for the performance of different approaches, we use the ground-truth data from White [3] and there are total 138 oracle links on the method-level. These links were manually identified by three different judges respectively, and then the disagreements were inspected collectively. In order to evaluate the performance of link establishing approaches, the following three measures are used: The first one is the "*Recall*" value, which is used for analysing how many expected links are successfully predicted, by a approach. The following equation is used to calculate the measures:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{\#OfValidPredicts}{\#OfGroundTruthLinks}$$

The next one is the "*Precision*", which is used to describe the relationship of how many predicted links are expected. The formula below is used to calculate the measure.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{\#ofValidPredicts}{\#ofAllPredicts}$$

"*F1-Scores*" is used to balance the value of the "*Precision*" and "*Recall*". The formular for "*F1-Scores*" is:

$$F1Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Project Name	Tested Path	Test Path
Commons-IO	src/main/	src/test/
Commons-Lang	src/main/	src/test/
JFreeChart	source/	tests/
Guava	guava/src/	guava-tests/
Apache-Ant	src/main/	src/tests/
ArgoUML	src/argouml-∗/src/	src/argouml-∗/tests/

Table 2: Paths for the tests and functions in six projects

Experiments Subjects. As to the first research question, we select the following six open-source Java projects on GitHub: *Guava*, *Argouml*, *Apache-Ant*, *Commons-Lang*, *Commons-IO*, *JFreeChart*. As for the second research question, the following three projects are used: *Commons-Lang*, *Commons-IO*, *JFreeChart*. We select these projects as our subjects because they are famous, the widely-used in many projects, continuously and frequently maintained on the GitHub website.

4.3 Experimental Setup

In the projects, we use various different parameters, so this section discusses all of them on each projects.

Pathes. In the experiments, the paths of source codes are used to distinguish the tests and functions. Table 2 shows the two paths for the six projects that are used in our experiments. In the four projects, *Commons-Lang*, *Commons-IO*, *JFreeChart* and *Guava*, all tests and functions are stored in two separate directories. In *Apache Ant*, it is slightly different, but the number of the tests and the functions in the two directories occupy about 95% of the total number. The most of the remained ones are those for the tutorials or the example codes etc, so it is reasonable to only consider the codes in the two directories. Regarding *ArgoUML*, the project structure is relatively-complex, because the test and the product codes are put into many different directories. Fortunately, the directories for both test codes and product codes follow two patterns. We demonstrate the two patterns in Table 2, in which the character ∗ is the wildcard.

Parameters. As to the Commits-Filtered Co-evolution approach, there are four filters in the and these filters base on the the values of the *Min*, $Q1$, $Q3$ and *Max* to the number of each types of changes in a project. Figure 4 shows the box-plot for the number of each type of changes in three projects, which also demonstrates all parameters that we used in the approach. Specifically, there are 12 boxes: the top four are for the "*JFreeChart*", the middle and the bottom boxes are for the "*Commons IO*" and "*Commons Lang*" respectively. The different colours means different types of changes: the black is for all changes, the red is for "ADD-METHOD", the blue is for "MODIFY-METHOD" and the green is for "RENAME-METHOD". Therefore, in the first two filters, we focus on the *IQR* and the length of whisker for black boxes in the diagram. For the other two, we find out the these values from the boxes in other three colours. As for the APRIORI algorithm, there are four parameters. The value of the $min\_confidence$ is universal for all projects, but the remained three are depended by the project. Table 3 shows their values. Specifically, the column 2 is the median for the number of commits in which at least one test was changed. The column 3 is for commits where at

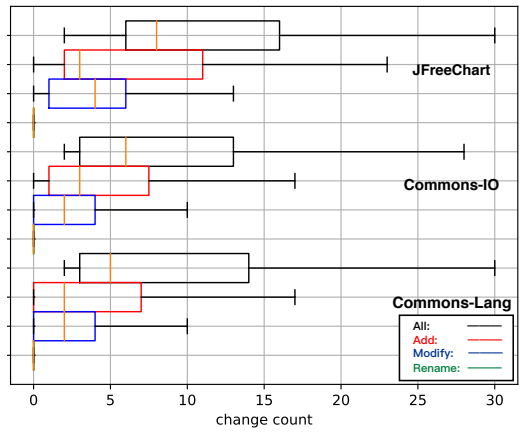

Figure 4: Changes to tests per commits

Projects	Functions	Methods	CoEvolution	Confidence
Commons-Lang	1	3	2	0.5
Commons-IO	1	3	2	0.5
JFreeChart	2	1	1	0.5

Table 3: Counts for the *co-evolved commits* and *evolved commits*

Projects_Name	*NoC_Test*	*NoC_Func*	*NoC_T&F*	*NoC_T‖F*	*T&F_Rate*
Commons IO	562	558	251	869	28.88%
Commons Lang	1342	1590	698	2234	31.24%
JFreeChart	287	1333	81	1539	5.26%
Guava	1830	2201	1022	3009	33.96%
Argouml	703	6153	147	6709	2.19%
Apache-Ant	289	4933	137	5085	2.69%

Table 4: Counts for the *co-evolved commits* and *evolved commits*

least one function was changed. The column 4 is for the commits where at least one test and one function changed.

4.4 RQ01: Tests Maintenance Behaviour Analysing

For the first research question, we investigated the test maintenance behaviours of the developers in six open source projects.

Commits Count. Table 4 demonstrates the counting results. Specifically, from left to right, the five columns record: *Projects_Name* [3]; *NoC_Test* [4]; *NoC_Func* [5]; *NoC_T&F* [6]; *NoC_T‖F* [7]; *T&F_Rate* [8]. According to the table, we obtain two new findings:

Finding01. *In half of the subjects, the changes to the tests are significantly less than that to the functions.* Especially in the Apache-Ant, the value of *NoC_Func* is even around twenty times as many as

[3]The name of each project
[4]The number of commits with at least one changed test
[5]The number of commits with at least one changed function
[6]The number of commits with at least one changed test **and** function
[7]The number of commits with at least one changed test **or** function
[8]The ratio of the commits when tests changed to those when codes changed (i.e. the value of column 4 divided by column 5 in each row).

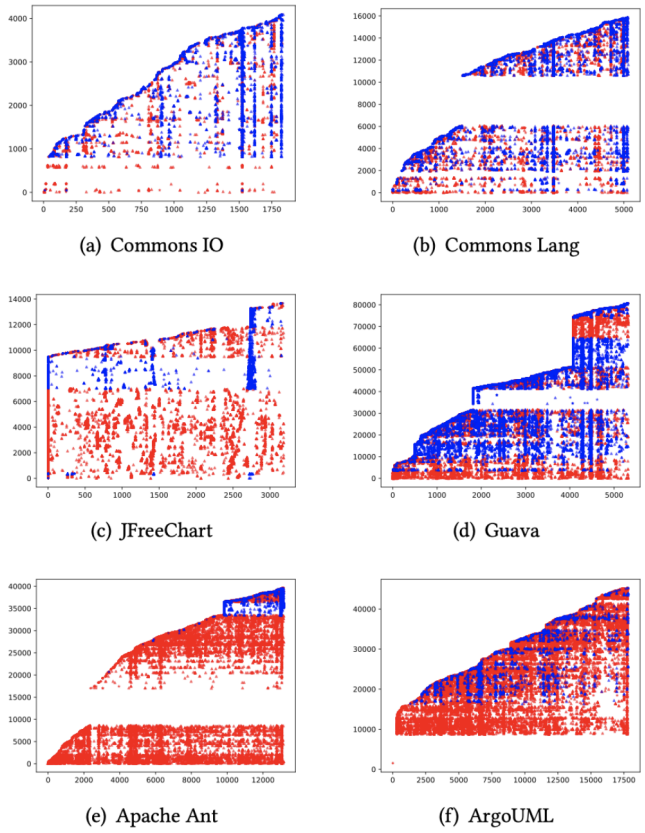

Figure 5: Change History View for six open-source projects

that of *NoC_Test*. The ratios are also about eight in the ArgoUML and JFreeChart.

Finding02. *In all subjects, the frequency of commits, where both of the tests and the functions were changed, is small.* It can be proven by the values of *T&F_Rate*. The figures are very low in all projects. In those three with frequent test maintenance, the value is only about 30%. Regarding to other three projects, the number is less than 5%.

Change History View. Figure 5 is the "*Change History View*" for the six project. The id of each method bases on their order of creation, and that of commit is also chronological. Our program removes all changes about deleted tests and functions from the database. Therefore, there will be a gap in the diagram if too many methods are deleted. One example is in the Common-Lang, whose developers stored the two versions of codes in the different directories of the same repository and deleted one of them in the latter commit. The diagram provides us two useful information:

Finding01. *The changes to the tests are significantly less than the changes to the functions in half of these subjects.* In "*Apache Ant*", most changes to the tests occurred in commits whose ids are larger than about 10,000. In the previous commits, the developers rarely create and never update the tests. The similar issues also occured in "*ArgoUML*" and "*JFreeChart*". It means that the tests of

Projects	Precision	Recall	F1 Score
Commons-Lang	0.43%	65.38%	0.87%
Commons-IO	0.22%	80.95%	0.45%
JFreeChart	0.02%	24.32%	0.04%

Table 5: Performance for the Co-evolution approach

Projects	Precision	Recall	F1 Score
Commons-Lang	8.06%	48.72%	13.84%
Commons-IO	8.15%	45.23%	13.82%
JFreeChart	0.02%	24.32%	0.04%

Table 6: Performance for the Co-creation approach

Projects	Precision	Recall	F1 Score
Commons-Lang	10.26%	5.12%	6.84%
Commons-IO	9.66%	33.33%	14.97%
JFreeChart	0.02%	10.81%	0.04%

Table 7: Performance for the APRIORI algorithm approach

Repository	Filter Range	Precision	Recall	F1 Score
Commons-Lang	[Min, Max]	20.30%	34.62%	25.59%
	[Q1, Q3]	35.71%	25.64%	29.85%
Commons-IO	[Min, Max]	16.05%	30.95%	21.13%
	[Q1, Q3]	21.43%	21.43%	21.43%
JFreeChart	[Min, Max]	20.00%	2.70%	4.76%
	[Q1, Q3]	20.00%	2.70%	4.76%

Table 8: Performance for the CoEv with a filter for the number of changes in each commit

Repository	Filter Range	Precision	Recall	F1 Score
Commons-Lang	[*Min*, *Max*]	35.84%	24.35%	29.01%
	[*Q*1, *Q*3]	33.33%	10.26%	15.69%
Commons-IO	[*Min*, *Max*]	22.22%	19.05%	20.51%
	[*Q*1, *Q*3]	30.43%	16.67%	21.54%
JFreeChart	[*Min*, *Max*]	20.00%	2.70%	4.76%
	[*Q*1, *Q*3]	20.00%	2.70%	4.76%

Table 9: Performance for the CoEv with a filter for the number of each type of changes in each commit

three widely-used and continuously-maintained projects are not frequently changed during developing.

Finding02. *In half of these projects, the modifications to the test codes are clustered in certain period instead of whole life-cycle.* One example is the commits of the "*JFreeChart*" with an id number about 2,700, where almost all tests changed. The similar patterns also occurred in the commits of "*Guava*" with id about the 4,300 and the commits of "*Common IO*" with id about 1,500 etc.

4.5 RQ02: Performance Evaluation

Co-evolution. Table 5 shows the result of the co-evolution approach on the three projects. It demonstrates the precision, recall, and f1 score. The result is similar to that in the previous research in classes level, in which the precision and recall are very low. Specifically, the precision values are less than 1% on all of the three projects. The recall values are more reasonable in the *Commons IO* and *Commons Lang*, but the figure is only 24.32% on the JFreeChart, which implies that only around a quarter of tests were co-evolved with their functions. Besides, the approach performs a relatively better result on the "*Commons IO*" and "*Commons Lang*" than on the "*JFreeChart*". Such differences show that the projects depend on the approach, and to be more specific, by the behaviours of the developers for maintaining the tests.

Co-creation. Similarly, Table 6 shows the values of the three measures for the three projects. As to the "*Commons IO*" and "*Commons Lang*", their precision increase from less than 1% to about 8%, but they are still very low. The values of recall moderately reduce to around 45%, and that of f1 score improve to about 14%. As for the "*JFreeChart*", it performs the same values for the precision, recall and f1 score on the three projects to that on the co-evolution approach, so the tests and the functions are only co-created in the project.

APRIORI Algorithm. Table 7 shows the performances of the approach, in which the precision, recall and f1 score is about 10% for precision in the "*Commons IO*" and "*Commons Lang*", the recall values are different in three projects. Besides, the approach performs the terrible result in the "*JFreeChart*", which is similar to other approaches.

Commits-Flitered Co-evolution. Table 8 shows the performance of the approach with two different general filters. Specifically, the "*Commons Lang*" and the "*Commons IO*" perform the similar precision and recall in about [15%, 35%]. And besides, when using the second filtering range, it performs better precision, recall and f1 score. Regarding the "*JFreeChart*", the values of measures when using the first range equal that when using the second one. The values for precision are 20%, but the recalls are only about 2% so that for the f1 score are also less than 5%.

Table 9 shows the performance of the approach with more specific filters on different types of changes. The result is mixed, and it shows better values for precision but lower values for the recall. The f1 scores for the two filters are similar. As to the "*JFreeChart*", it is completely same with previous.

5 RELATED WORK

Zaidman[10] has investigated the similar research question about the co-evolution relationship for the test methods and tested functions, in which they used "*Change History View*", "*Growth History View*" and "*TestCoverage Evolution View*" to analyse whether the methods and functions are evolved synchronously or in phases. Their paper mined the repositories from *CVS*, but our research question focuses that from *GitHub*. And besides, they did not analyse the projects that we used in the experiments. Therefore, the outcomes of this paper cannot completely address our first research question.

Levin [13] has implemented a tool , *CodeDistillery*, which is also able to traverse a Git repository and to identify the different types of changes in a file. The main difference between the two tools is their purpose: Their tool focuses on the repository mining and

the changes classifying; By contrast, our tool more cares about the test-to-code traceability links. There also certain differences in the design and implementation of the two tools.

Rompaey [7] has implemented a similar idea for test-to-code link establishing by the CoEv approach. Their works focus on the classes-level, and their purpose is to compare the performance for different approaches. By contrast, our work implements that idea on the methods-level with three optimisations, and the research purpose focuses on the feasibility for the co-evolution approaches.

White[3] analyses several different links establishing approaches on both the method-level and the class-level. The results of these approaches are used to calculate a score. After that, they compare these scores with a predefined threshold to find out the possible links. Our works use the ground-truth oracle links from White's works. Compared with the predicted links from the TcTracer, those from our works show relatively low precision and low recall on the method level in three open-source projects.

6 CONCLUSION

In this report, we analyse the test maintenance approach used in six open-source Java projects. We find that most of them use the asynchronous maintenance approach, and the developers of these projects change a large number of methods in continuous commits occasionally, which indicate the unreliability of our previous assumption for co-evolution between the test methods and tested functions.

we present an tool for establishing test-to-code traceability links between the test methods and tested functions. In the tool, we implement the CoEv approach, Co-Creation approach, Commits-Filtered CoEv approachand the APRIORI algorithm on method-level. Besides, it is able to mine the repository and classify three different type of changes on the method-level.

After evaluating the performances of these approaches on the method-level to three open-source Java projects, we find the CoEv approach perform with low precision and recall. The values of the two measures slightly increased after optimising, but the best result of the precision and recall are still only about 35% and 24% respectively.

REFERENCES

[1] Jane Huffman Hayes, Alex Dekhtyar, and David S. Janzen. Towards traceable test-driven development. In *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE 2009*, pages 26–30. IEEE, may 2009.

[2] Abdallah Qusef. Test-to-code traceability: Why and how? In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–8. IEEE, dec 2013.

[3] Robert White, Jens Krinke, and Raymond Tan. Establishing Multilevel Test-to-Code Traceability Links. In *Proceedings of the International Conference on Software Engineering*, 2020.

[4] Serge Demeyer. Object-Oriented Reengineering. In *Software Evolution*, pages 91–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] ISO. ISO/IEC 15504-5 Information technology – Process assessment – Part 5: An exemplar software life cycle process assessment model. 2006.

[6] H.M. Sneed. Reverse engineering of test cases for selective regression testing. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, volume 8, pages 69–74. IEEE, 2004.

[7] Bart Van Rompaey and Serge Demeyer. Establishing Traceability Links between Unit Test Cases and Units under Test. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 209–218. IEEE, 2009.

[8] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying Fine-Grained Co-evolution Patterns of Production and Test Code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, sep 2014.

[9] Laszlo Vidacs and Martin Pinzger. Co-evolution analysis of production and test code by learning association rules of changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 31–36. IEEE, mar 2018.

[10] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, jun 2011.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[12] Rakesh Agrawal and Ramakrishnan S&ant. Fast Algorithms For Mining Association Rules In Datamining. *International Journal of Scientific & Technology Research*, 2(12):13–24, 2013.

[13] Stanislav Levin and Amiram Yehudai. Processing Large Datasets of Fined Grained Source Code Changes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 382–385. IEEE, sep 2019.

ACCESS TO SOURCE CODE

The code and most resources used in the paper are stored in the GitHub repository