INTERNATIONAL JOURNAL OF COMPUTERS
DOI: 10.46300/9108.2023.17.2 Volume 17, 2023

Visualization of Source Code and Acceptance Test Co-evolution

Ali Görkem Yalçın, Tugkan Tuglular
Izmir Institute of Technology,
Urla, Izmir, 35430,
Turkey

Received: January 2, 2023. Revised: September 4, 2023. Accepted: October 22, 20223. Published: November 23, 2023.

Abstract—This research investigates how the codebase and acceptance tests have changed over time using version control. An essential requirement for this evaluation is analyzing software development and test cases used in real-world projects. The data stored in software repositories is vital for software systems. Access to prior versions of the program and the ability to differentiate between adjacent versions using acceptance tests and source code modifications can provide insight into how the software evolves. A skillful visualization method is required to achieve this level of awareness. This research presents a visualization method for tracking the concurrent evolution of source code and acceptance tests within software development projects. The case study looks at four open-source projects taken from the real world that have acceptance tests hosted on GitHub. The proposed visualization method enables the negative slope on two projects for the source code and acceptance test co-evolution to be quickly observed, and with such an observation, the project team can take necessary actions to change the course of the co-evolution.

Keywords—Mining software repositories, Software metrics, Software test co-evolution, Software visualization.

I. Introduction

VISUALIZATIONS can shed light on several facets of the software development process, and the combination of multiple visualization approaches can provide a more comprehensive understanding of the evolution of the codebase under consideration. This paper proposes a visualization technique for software projects with the necessary attributes to monitor source code and acceptance test co-evolution.

Visualizing the progress of source code can aid in understanding the growth of a codebase. It is possible to visualize the development of source code through file version history. Many version control systems, such as Git, monitor the history of individual file modifications. Viewing the history of a specific file can give insights into when changes were made.

Examining the progress of source code in conjunction with the advancement of acceptance tests offers a comprehensive perspective on developing a software project. The development of source code and acceptance tests can be systematically traced to demonstrate a definitive correlation between modifications made to the code and corresponding adjustments in the acceptance tests. This tracing can help us comprehend how code modifications correspond to changes in the software's functionality-validating tests. This traceability can be utilized to visualize the relationship between code modifications and the corresponding acceptance tests. This traceability allows us to identify patterns, such as whether code modifications result in test updates or the creation of new tests. In addition, we can see how acceptance tests have been updated to account for new features, modifications, and bugs.

This study presents a novel visualization method that facilitates the correlation between acceptance test modifications and corresponding software code changes. Consequently, it allows for the comprehensive documentation of the program's evolution and the associated acceptance tests. This approach guarantees that the project team possesses comprehensive visibility into the co-evolution of the source code and acceptance tests. This observability enables the team to undertake appropriate measures to enhance the quality of the product and ensure that it aligns with user expectations as it progresses.

We concretized the proposed approach by using data mined from GitHub repositories. As explained in Section 4, the project selection resulted in 21 projects with acceptance tests. We take the top four projects with maximum major or minor version counts for the proposed visualization technique.

The structure of this paper is as follows. The second section is a summary of the relevant literature. The third section explains the proposed method and describes the instrument that produced the deliverables. The results and analysis of this investigation are presented in Section 4. The paper concludes with Section 5.

II. RELATED WORK

Software development involves more than adding features. It also requires testing. The software evolves with each release. Software and test suites should evolve together. If test suites are not updated for each program change, they become old and useless. Tests are sensitive to even the slightest changes in the code, [1]. The software's functionality can be altered by modifying a boolean variable. Some tests may become irrelevant due to these modifications, leading to untested production code. According to [1], a 1% change in program branches might result in a 16% decrease in test code coverage. The decay in testing results from fixing bugs, modifying existing features, adding new features, and reworking existing code. As a result, updated application test suites must be patched or regenerated, [2], [3], [4], [5], [6]. Fixing test cases for every program version can be time-consuming, [7]. It is essential to comprehend the how and why of test repair for the process of test evolution, [8].

The study, [4], studied the co-evolution of test and production code throughout the software's lifecycle, utilizing versioning system software repositories. The primary area of emphasis for their research was unit testing. The study utilized statistical methods to analyze many aspects, such as production and test code revisions, file changes for each version, and software test coverage. The researchers discovered the data mentioned above on two open-source projects and one industrial project, enabling them to ascertain the degree of synchronicity between the evolution of test and production code. Additionally, they could identify whether there was an increase in test writing activity before significant software releases. In addition, they proposed the utilization of visual aids to monitor the progression of software testing and the evolution of production code. It was discovered that the implementation of development updates initiated the evolution of the project in two out of three endeavors. In the other project, there was a more simultaneous evolution of both the test and production code. The rate of evolution has never surpassed the rate of production code development. Before a major release, these projects lacked a substantial testing phase for writing tests. Similar to previous studies conducted on this subject matter, a notable limitation of this study is its analysis of only three projects. Consequently, the findings derived from this limited sample size may not apply to a broader population but only to those projects that align with the characteristics of the studied cases.

In, [9], the authors presented a methodology aimed at the automated restoration of unit tests that have become outdated due to alterations in method declarations or signatures. The primary area of emphasis was on outdated unit test cases that resulted in compilation issues. The researchers analyzed 262 iterations of test and production code evolution during the problem-solving process across 22 open-source projects. Out of the 22 projects now under review, it was observed that 53% of the versions did not update the method signature. Consequently, the test methods associated with these versions produced no compilation issues. The primary area of their investigation revolved around test repair. However, throughout the development and implementation of the repair tool, they analyzed the program through the process of repository mining, examining the co-evolution of both test and production code.

The study, [7], initially examined 80 iterations of software systems to assess the nature of modifications made by developers when updating outdated test cases and how developers repurposed existing test cases to build new and accurate ones. The researchers extracted data from the software repositories to analyze the program modifications between different versions and obtain relevant information. Their investigation primarily focused on the disparities in method signatures across different versions. Based on the provided information, the researchers could ascertain whether the test case for the specified technique had been modified and, if so, the modifications' nature.

According to, [5], it is imperative to comprehend the evolution of test cases in real-world projects to facilitate the automated repair of such test cases. Without practical obstacles, the acquired knowledge remains applicable only within a limited scope of test cases or software projects. The study incorporated a total of six real-world projects alongside the implementation of unit testing. The test suite evolution of these projects was classified into four categories: test repairs, additions, removals, and refactoring. Statistics regarding the test changes were furnished based on six real-world initiatives. The tests that have been refactored and corrected are called Test Modifications. The process of modifying a deprecated test case and successfully executing it is commonly referred to as test repair. 6.4% of the modifications made to the test were classified as fixes. During the process of test refactoring, the test undergoes modifications that involve incorporating a new library and altering variable names. 22.9 percent of the test modifications conducted were categorized as refactoring. The test deletions encompassed outdated, challenging-to-rectify, and unnecessary assessments. Despite the discontinuation of the test, its functional aspects continue to persist in the projects, posing challenges in rectification. The test cases were rewritten entirely due to the perception that they were too intricate to rectify.

The study, [10], developed a software application called TestEvol. This tool was specifically built to analyze the evolutionary patterns of test suites for Java programs and their associated JUnit test cases. The tool developed by the authors can analyze two distinct versions of a program with their corresponding test suites. The purpose of this system is to detect any changes made to the unit tests and generate statistical information regarding the influence of these adjustments on the code coverage.

The study, [11], has developed a software tool that aids in examining specific software versions, their associated unit tests, and the progression of code coverage. The technology was utilized in six pragmatic applications, resulting in

statistical data from these endeavors. The tool provided by the authors presents information regarding the co-evolution of unit tests and source code, the size of software patches concerning patch type, and statistics on code coverage.

The analysis undertaken by, [12], focused on five open-source projects to investigate the co-evolution trends between unit tests and source code components. Their approach's novelty is in applying associative rule mining techniques to detect complex co-evolutionary associations. A comprehensive analysis of the five studies revealed the presence of a total of 12 unique patterns, some of which were seen across multiple projects. A qualitative analysis was performed to investigate the statistical data related to the incorporation of the test class alongside the source class within the same commit, the subsequent commit without the test class, or the alteration of the existing test class resulting from adding the new source class.

The study, [13], also investigated the co-evolution relationship between source and unit test codes. The research encompassed a comprehensive set of 61 open-source projects and analyzed a substantial dataset of over 240,000 contributions from well-known open-source project platforms such as GitHub and BitBucket. A notable finding indicates that test maintenance was noticed in more than 50% of the changes in certain projects. In specific cases, the proportion was less than 15%. The proportion of commits allocated to the maintenance of tests in the absence of a particular project surpassed 68.5%. Additionally, it is essential to acknowledge that some projects exhibited a deficiency in having a specialized maintenance system for conducting tests, leading to a dependence on the project for testing.

The investigation carried out by, [6], analyzed the progression of unit test suites in eight distinct iterations of Java systems. The primary aim of this study was to acquire a deeper understanding of the procedures and methodologies utilized in developing and refining these test suites. The authors saw a rise in the size of the test suite over time, as indicated by their research findings. Additionally, it was observed that the complexity of the tests stayed consistent throughout the software's progress. Additionally, it was discovered that the efficacy of the test suite generally increased as time progressed. Regarding code coverage, it exhibited a rise in 45.7% of the software versions, maintained stability in 28.6% of the versions, and experienced a decrease in 25.7%.

All the above research is merely concentrated on source code and unit tests. The authors conducted the first research on the source code and acceptance test co-evolution, [14]. This paper builds on that research by adding a visualization approach to demonstrate the co-evolution. Moreover, previous research on source code and unit test co-evolution has not considered the visualization-related metrics but on tool development for the examination of co-evolution. All the values in the above literature would have been visualized even over time, not as a snapshot.

III. Proposed Method

Following the metrics at the project level proposed by, [15], our approach collects the source code's size and the addition and removal of source code as a line of code. On top of that, we added the size of the acceptance tests and the addition and removal of acceptance tests as a line of tests. Unlike, [15], we propose presenting these values with respect to versions on a version count line in increasing order. This way, we enable developers, team leaders, and project managers to observe the growth and stagnation phases in the software evolution. The same is valid also for acceptance tests. Moreover, by visualizing both source code and test code progress simultaneously, the co-evolution can be observed, and necessary decisions can be made for project management. Visualization of the co-evolution of source code and acceptance gives clues about the quality of the software, which we believe to be very important.

We also propose to visualize the addition and removal of source code and acceptance tests as a scatter plot. This way, we enable developers, team leaders, and project managers to observe changes and especially improvements version by version. In our approach, we include version information concerning semantic versioning, which is categorized as major, minor, and patch versions. Table I shows our count metrics, [14]. Furthermore, we developed ratio metrics shown in Table II, [14].

Table I. Count metrics for the software projects, [14].

Acronym	Explanation
VC	all Version Count (incl. patches)
MMVC	Major-Minor Version Count
SLOC	Source Line of Code
TLOS	acceptance Test Line of Scenario

On the scatter plot graphs, we also provide a trend line for the MMTOA metric to indicate the trend of how acceptance tests have co-evolved with source code and give an idea about the expectance of future versions. The software development team can, of course, reverse the trend. To achieve that, they need to know, and visualization summarizes it well.

Spider charts, often referred to as radar charts, are helpful as a visualization tool for simultaneously representing and comparing several aspects of data. Providing a comprehensive perspective of numerous metrics can assist in concretizing the strategy of code evolution and acceptance test evolution. Spider charts offer a visually captivating and intuitive method for displaying multidimensional data, rendering them well-suited to analyze the progression of code and acceptance tests across our ratio metrics.

Table II. Ratio metrics for the software projects, [14].

Acronym	Explanation	acceptance Test Update	Source code Update
MMTOA	for Major-Minor version types, acceptance Test updates / (Over) All updates	Always	Don't care
MMSOA	for Major-Minor version types, Source code updates / (Over) All updates	Don't care	Always
AVOTOA	For all Version types, Only acceptance Test updates / (Over) All updates	Always	Never
AVOSOA	For all Version types, Only Source code updates / (Over) All updates	Never	Always
AVSATOA	for All Version types, Source code, And Test updates / (Over) All updates	Always	Always
AVTOA	for All Version types, acceptance Test updates / (Over) All updates	Always	Don't care

Progress is tracked over versions through the spider charts, enabling developers to make well-informed judgments regarding enhancements to the source code and the acceptance tests. It is important to note that spider charts should be utilized alongside other visualization approaches to comprehend the development process.

IV. Case Study

We selected real-world open-source applications from GitHub for our analysis. Our first project selection criteria was identifying projects containing ".feature" Gherkin files. We found all projects with stars and Gherkin files using GitHub's proprietary search engine using the search query: "stars:>1 language: gherkin". We identified 601 projects that met these criteria. Using scraping, we then obtained the name, URL, stars, versions, files, and percentage of each programming language type in the project.

We filtered the data set for projects having at least two versions. At least two versions are necessary to track project evolution. Filtering decreased from 601 projects to 146. We then examined each project version for Gherkin file updates. If a project does not update its Gherkin files, acceptance tests do not evolve. We eliminated those non-qualifying projects. Next, we eliminated projects without Gherkin file step definitions, reducing projects to 61. We eliminated projects with fewer than five major-minor versions from those remaining 61, leaving 23. Finally, outlier filtering excluded two more projects from the remaining 23, leaving 21. We used extensive project graphs and charts in our investigation. The top four projects with the most major or minor versions are chosen. The project names and acronyms are in Table III, and their short explanations are given below:

1) BEWMCHS is a plugin project for a project called Behat, which is an open-source Behavior Development framework for PHP.
2) FBR is a framework for defining and using factories instead of fixtures.
3) JEFTBS enables the formatting of bibliographies and reading lists and eases the process of citation insertion.
4) TSCLI is a tool for managing parallel versions of multiple SDKs (Software Development Kits).

Table I and Table II demonstrate the ten attributes we utilized throughout the study to characterize the projects we chose to analyze. We used JSoup with Selenium to scrape the data for those attributes for each version of each project. Data was extracted for files that were modified in each version. Regarding the revised files, we analyzed them to determine the number of line additions and deletions, as well as the quantity of source and test code lines that were added and removed.

Table III. Acronyms and names of the selected projects.

Acronym	Project Name
BEWMHCS	Behat extension with most custom helper steps
FBR	Factory Bot Rails
JEFTBS	Jekyll extensions for the blogging scholar
TSCLI	The SDKMAN! Command Line Interface

Additionally, we tracked the number of specific gherkin keywords, including Given, Then, When, and Scenario, that were added or removed. With the scraped data in hand, we built a network of charts to illustrate the interdependencies across different projects. Each of the four projects had its own set of spider charts and scatter plots. An explanation of each of these charts follows.

A. Scatter Plots Representing Co-evolution

A scatter plot employs individual data points to illustrate the relationship between two distinct numerical variables visually. The spatial location of each point on a graph represents the numerical magnitude of an individual data point. Based on the obtained outcome, doing a correlation study or identifying an outlier is possible. Scatterplots are mainly employed to observe and illustrate the associations between two numerical variables. We used scatterplots to display the cumulative total of ATLOC, a cumulative total of SCLOC, differential ATLOC, and differential SCLOC values as shown in Figure 1.

A trend line chart displays the changes in value that happen over time. We used trend line charts to display the test code update count / all code update count ratio for major and minor versions. These charts can display the ratio of test writing percentage for each project. We added a curve fitting to both display and estimate the project's test evolution.

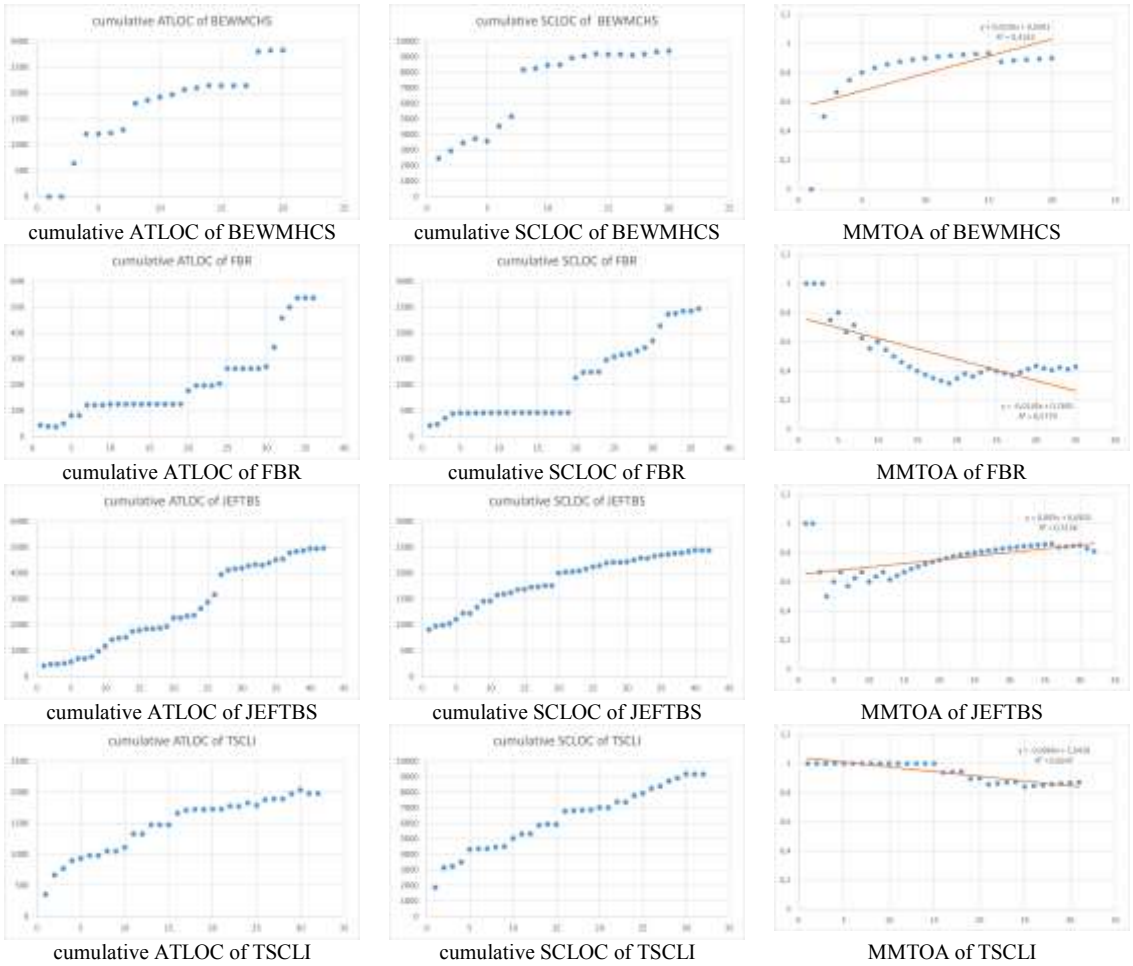


Fig. 1 Scatter plots of cumulative ATLOC and SCLOC for MMV and MMTOA trend of the four selected projects.

These graphs have both negative and positive slopes. The reason for that is if the first few versions contain test updates, then the curve fitting starts from 1 and converges to the final value, which results in a negative slope, and if the first few versions do not include any test updates, then the curve fitting start from 0 and converges to the final value, thus resulting in positive slope.

Curve fittings can be constructed with the help of these points. The straight lines with their respective equations and R^2 values constitute the fitting curve for this study's projects, as evident from the graphs below. The test-to-all ratio of upcoming projects and versions can be determined using this curve through prediction.

The difference between the two MMVs is suitable for the increment size estimation in differential meaning. In Figure 2, we can see that in most of the projects, the size of the ATLOC updates is smaller than that of the SCLOC updates. The main reason is that it can take a few lines of acceptance test code to test a functionality that takes hundreds of lines of code to implement.

B. Spider Charts Representing Co-evolution

Data with multiple properties can be visualized using spider plots. Since each of the four projects in the case study has its own set of four characteristics, this usage applies to the data presented there. These attributes are as follows: MMTOA, AVTOA, AVSATOA, and AVOSOA. To display the spider charts precisely, these values have been normalized between 0 and 1.

In the graphs shown in Figure 3, we displayed a project's course throughout its lifetime to see how it evolved for the above attributes. For each project, we created four intervals containing an equal number of versions, and we created a spider graph for each of the four intervals.

1. BEWMHCS From the first graph, we can see that the blue graph displays the spider chart in version 2, the orange graph displays the spider graph in version 9, the gray line displays the spider graph in version 16, and finally yellow graph displays the spider graph at version 22. AVTOA attribute starts as 0.5, the result of the first two versions containing one test update, but converges to 0.8, which can be seen from the remaining graphs.

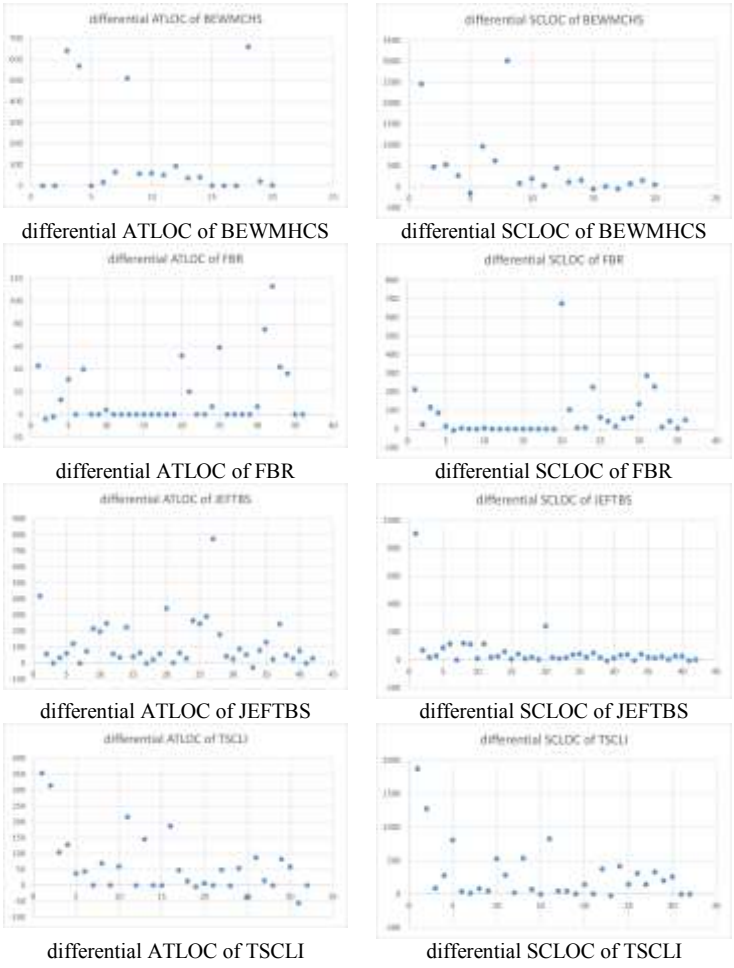


Fig. 2 Scatter plots of differential ATLOC and SCLOC for MMV of the four selected projects.

The AVOSOA attribute starts at 0.5 and slowly decreases to around 0.18. This decrease means that around 18% of the updates contain only source code, not test code, and this 18% value is consistent throughout the project's lifetime. The AVSATOA attribute starts at 0.5 since the first versions introduced source and test code updates or only source code updates. Then, this value converges to around 0.8. This convergence means that around 80% of the updates contain both source and test code, and this 80% value is consistent throughout the project's lifetime. MMTOA attribute value starts at 0.5 and converges to around 0.9. This convergence displays that tests get updated in major or minor updates, but tests rarely get updated in patch updates.

2. FBR As can be seen from the second graph, the blue graph displays the spider graph at version 2, the orange graph displays the spider graph at version 19, the gray line displays the spider graph at version 36, and finally, the yellow graph displays the spider graph at version 51. The AVTOA attribute starts as 1, the result of the first two versions containing test updates but drops down to 0.4 and lingers there for the remainder of the project's lifetime.

The AVOSOA attribute starts at 0 and slowly increases up to around 0.5. This increase means that around 50% of the updates contain only source code, not test code, and this 50% value lingers between 0.7 and 0.5 throughout the project's lifetime. The AVSATOA attribute starts at one since the second version introduced source and test code updates. Then, this value fluctuates between 0.291 and 0.45. This fluctuation means that around 45% of the updates contain source and test code. MMTOA attribute value starts at one and converges to around 0.42. The heavy drop from 1 to 0.42 resulted from the first two versions containing test updates, but after these initial versions, the lack of test updates resulted in this value being much lower.

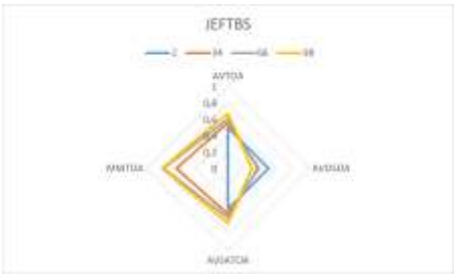
3. JEFTBS As can be seen from the third graph, the blue graph displays the spider graph at version 2, the orange graph displays the spider graph at version 34, the gray line displays the spider graph at version 66, and finally yellow graph displays the spider graph at version 98. AVTOA attribute starts as 0.5, the result of one of the first two versions containing test updates, and converges to 0.673.

The AVOSOA attribute starts at 0 and slowly increases up to around 0.295. This increase means that around 30% of the updates contain only source and not test codes. The

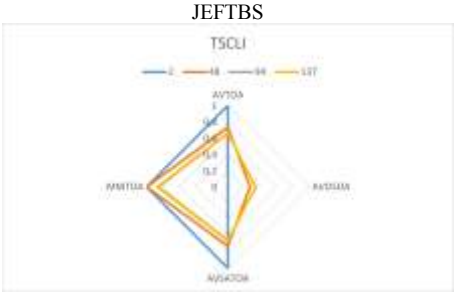
AVSATOA attribute starts at one since the second version introduced source and test code updates. Then, this value converges to around 0.673. This convergence means that around 67% of the updates contain both source and test code, and this 67% value is consistent throughout the project's lifetime. MMTOA attribute value starts at 0, the result of the first few versions not containing any test updates, and converges to around 0.8.

BEWMHCS

FBR



JEFTBS



TSCLI

Fig. 3 Spider charts of the four selected projects.

4. TSCLI. As seen from the last graph, the blue graph displays the spider graph in version 2. The orange graph displays the spider graph at version 48, the gray line displays the spider graph at version 94, and finally, the yellow graph displays the spider graph at version 137. AVTOA attribute starts as 1, the result of the first two versions containing test updates but drops to 0.7 and lingers there for the remainder of the project's lifetime.

The AVOSOA attribute starts at 0 and slowly increases up to around 0.4. This increase means that around 40% of the updates contain only source code, not test code, and this 40% value is consistent throughout the project's lifetime. The AVSATOA attribute starts at one since the second version introduced source and test code updates. Then, this value converges to around 0.6. This convergence means that around 60% of the updates contain both source and test code, and this 60% value is consistent throughout the project's lifetime. MMTOA attribute value starts at one and converges to around 0.8. This convergence displays that tests get updated in major or minor updates, but in patch updates, tests rarely get updated.

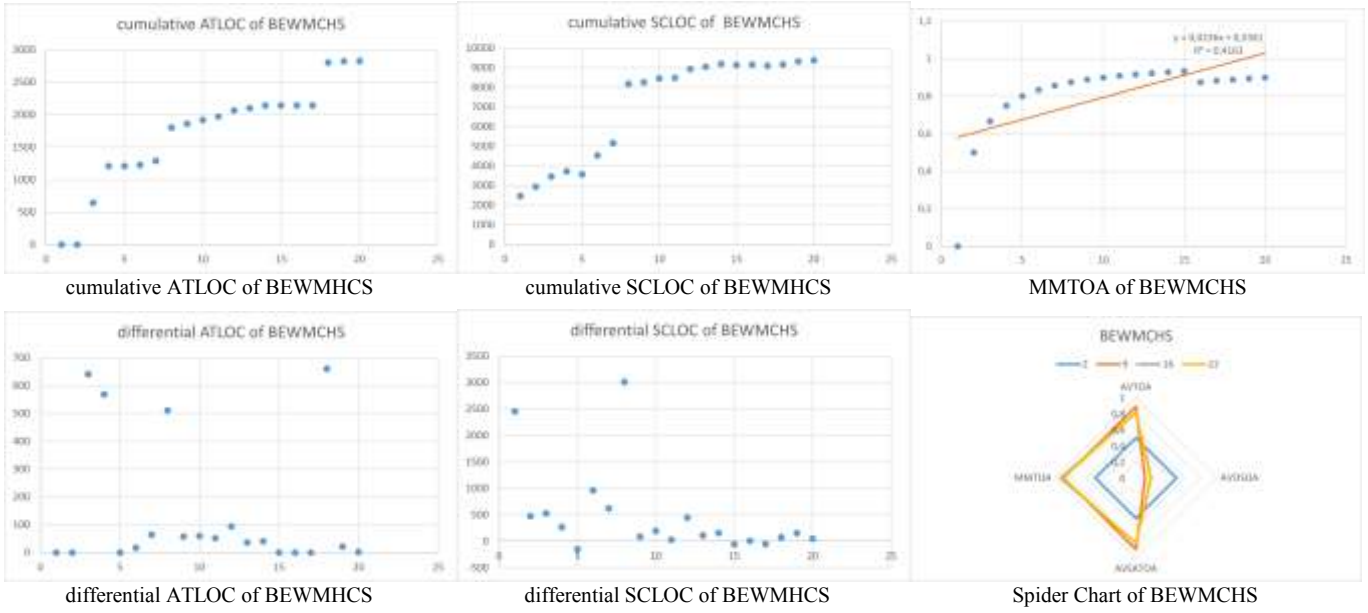
C. Dashboards Representing Co-evolution

This section displays a project dashboard that offers all the graphs and charts explained throughout the study. These dashboards aim to consolidate the analysis of the studied projects and their corresponding metrics into a single visual representation. This single representation allows for a holistic view of the analyzed projects. For instance, in the BEWMCHS project's dashboard in Figure 4, we can see that the differential ATLOC chart has a few (0, n) data points, but around 75% of the time ATLOC value is non-zero. These zero data points are not adjacent, which results in the MMTOA chart having a positive slope.

Similarly, we have many adjacent versions of the FBR project in Figure 5, where the differential ATLOC chart has many (0, n) data points, most in all projects. This results in the MMTOA chart having a negative slope. Also, in the spider chart of the FBR project, we can see that the MMTOA attribute is 1 for the first graph, and for the remaining spider graphs, the MMTOA value is much lower, thus justifying the first two graphs. These dashboards allow for the identification of similar trends, patterns, or relationships.

In the JEFTBS project, as shown in Figure 6, the differential ATLOC chart has few (0, n) data points. Similarly, the differential SCLOC chart has few (0, n) data points. That means that the co-evolution of source code and acceptance tests has happened in this project. The MMTOA chart having a positive slope supports this observation. The spider chart for the JEFTBS project in Figure 6 has similar shapes for the project's later phases, which means steady software development, including acceptance tests, took place.

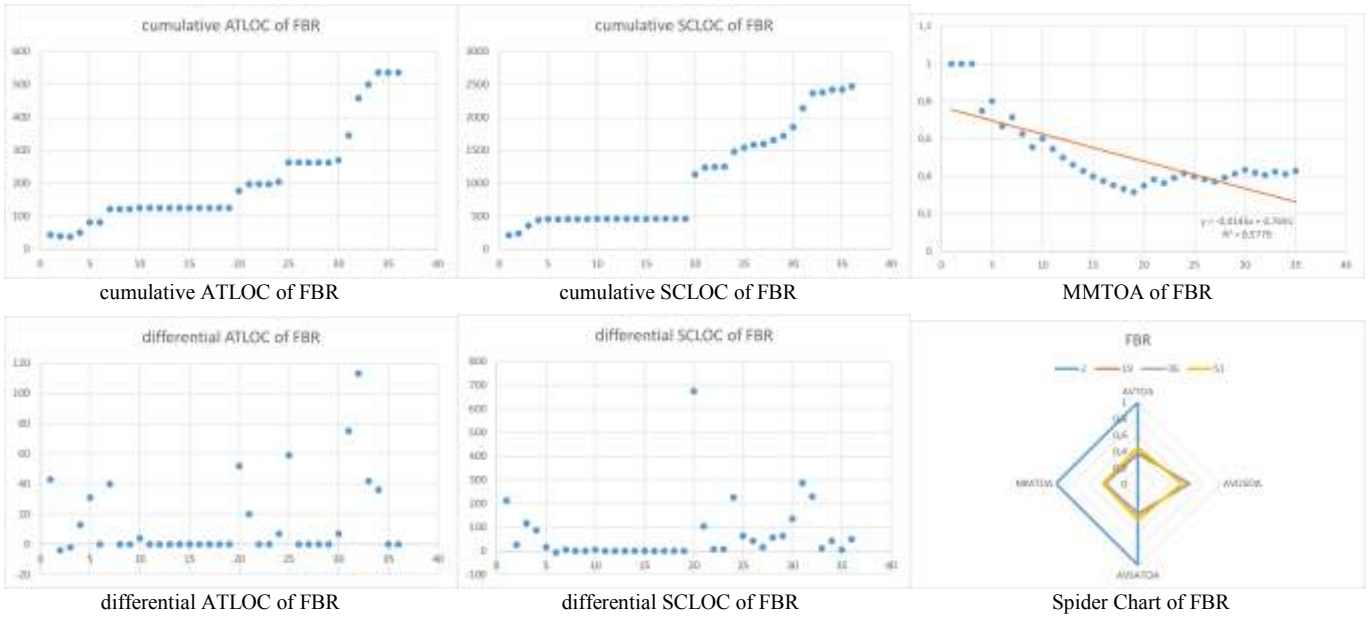
Figure 7 illustrates the dynamics of the TSCLI project in terms of source code and acceptance evolution. It would be simple to conclude that co-evolution does not exist based on the negative slope of the MMTOA chart. However, in contrast to the BEWMCHS project, the differential SCLOC chart and the ATLOC chart for the TSCLI contain a limited number of data points (0, n). These values signify that source code and acceptance test development is ongoing. Consolidating all the charts provides a more comprehensive narrative.



cumulative ATLOC of BEWMCHS | cumulative SCLOC of BEWMCHS | MMTOA of BEWMCHS

differential ATLOC of BEWMCHS | differential SCLOC of BEWMCHS | Spider Chart of BEWMCHS

Fig. 4 Dashboard of the BEWMHCS project.



cumulative ATLOC of FBR | cumulative SCLOC of FBR | MMTOA of FBR

differential ATLOC of FBR | differential SCLOC of FBR | Spider Chart of FBR

Fig. 5 Dashboard of the FBR project.

V.CONCLUSION

This research uses the presented visualization method to comprehend software evolution dynamics, particularly source code modifications and acceptance tests. This study examines software repository historical data to understand code update patterns and acceptance test tweaks, providing a fresh perspective on software development techniques.

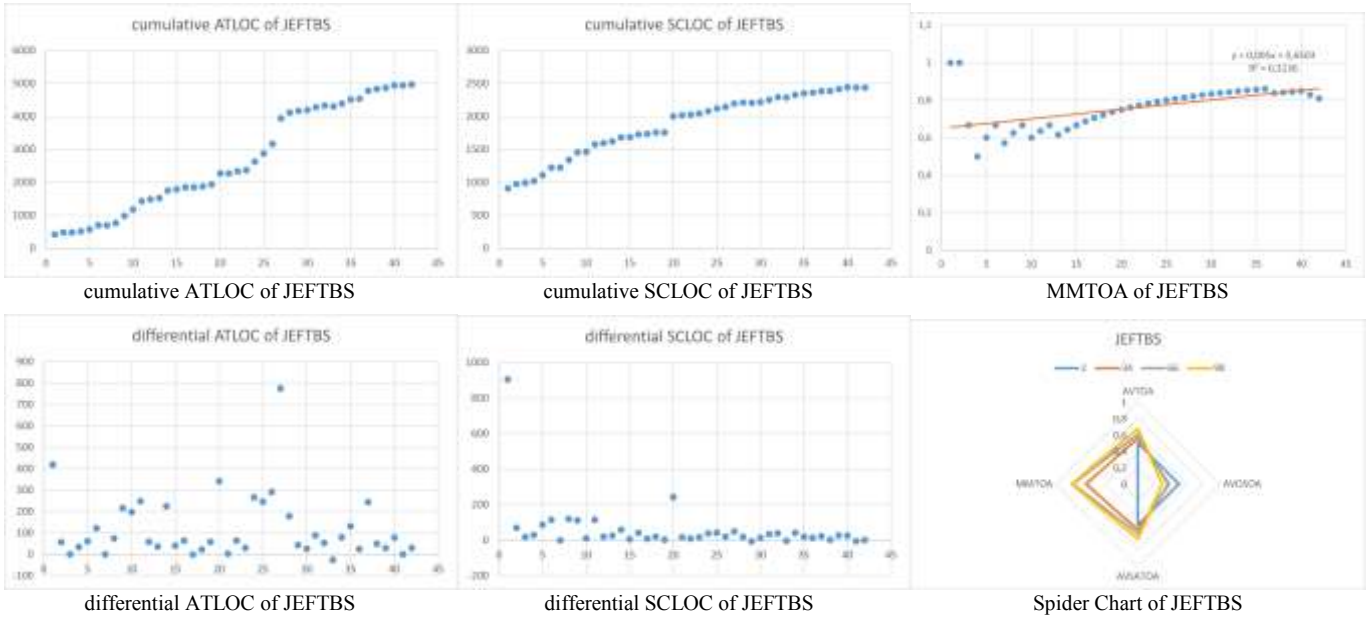
This research extracts and analyzes software repository data, focusing on the evolution of source code and acceptance tests. Advanced algorithms read and interpret codebase and test changes to find trends, anomalies, and essential development practice changes.

The visualization technique developed in this study is easy but comprehensive, allowing users to track software project evolution readily. It graphically shows how source code changes affect acceptance testing. This method helps understand the development process and identifies ways to improve software development techniques.

Four open-source projects on GitHub are used in the case study to prove that the visualization strategy works. These projects were chosen for their active acceptance testing and distinct sizes and domains, providing different data for study. The study highlights the practicality of the visualization method in different projects.

This analysis attempts to advance software engineering by revealing code and testing co-evolution best practices. It aims

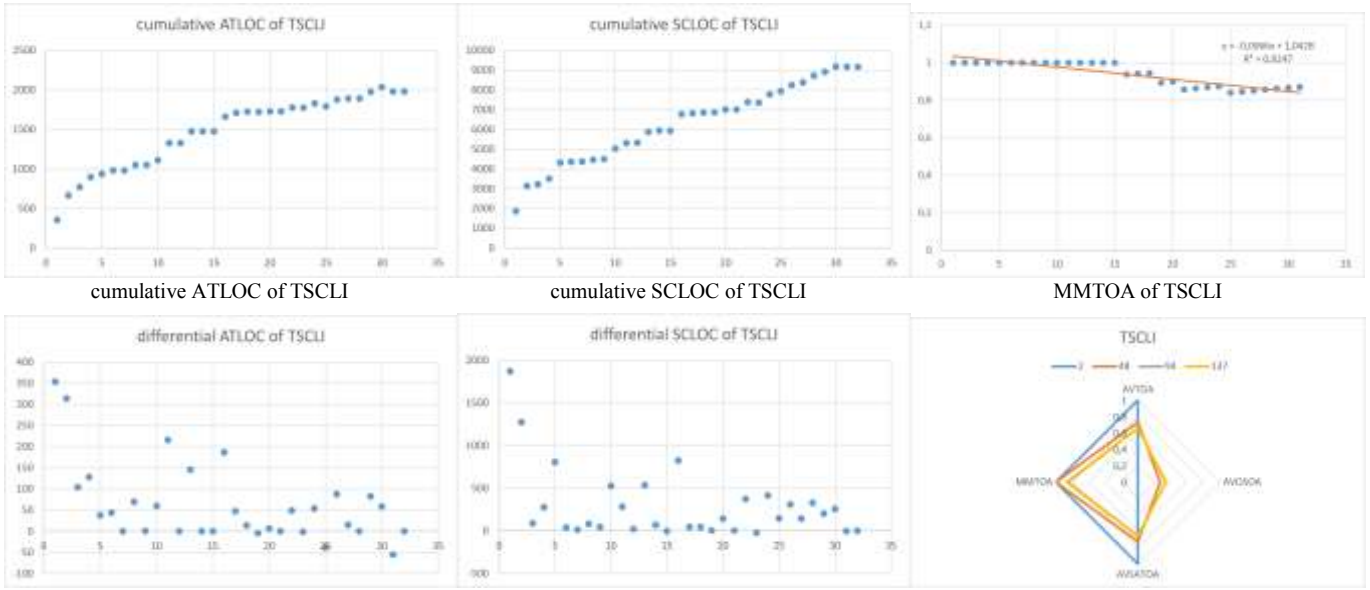
to give software developers and project managers a solid tool to manage the development process better.



cumulative ATLOC of JEFTBS | cumulative SCLOC of JEFTBS | MMTOA of JEFTBS

differential ATLOC of JEFTBS | differential SCLOC of JEFTBS | Spider Chart of JEFTBS

Fig. 6 Dashboard of the JEFTBS project.



cumulative ATLOC of TSCLI | cumulative SCLOC of TSCLI | MMTOA of TSCLI

differential ATLOC of TSCLI | differential SCLOC of TSCLI | Spider Chart of TSCLI

Fig. 7 Dashboard of the TSCLI project.

This research emphasizes the relevance of integrated development approaches in creating robust and dependable software by linking code modifications and acceptance testing.

We want to add unit tests to the co-evolution visualization technique in future research. This extension would improve the visibility of the software project development lifecycle. Unit tests are essential to software development and provide thorough insights into component functionality. These tests, acceptance tests, and source code updates help us comprehend the software's micro and macroevolution.

By incorporating this dimension into our study, we aim to provide a valuable contribution to the area of software engineering. This contribution will manifest as a comprehensive tool emphasizing various facets' interdependence within the software development process. This advanced visualization technique would facilitate the implementation of more streamlined, productive, and superior software development methodologies across a wide range of project settings.

References

[1] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings IEEE International Conference on software maintenance. ICSM 2001*, 2001, pp. 170–179.

[2] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting Repairs for Broken Unit Tests," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand: IEEE, Nov. 2009, pp. 433–444. doi: 10.1109/ASE.2009.17.

[3] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th International Symposium on Software testing and analysis - ISSTA '10*, Trento, Italy: ACM Press, 2010, p. 207. doi: 10.1145/1831708.1831734.

[4] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[5] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, Cary, North Carolina: ACM Press, 2012, p. 1. doi: 10.1145/2393596.2393634.

[6] N. Alsolami, Q. Obeidat, and M. Alenezi, "Empirical analysis of object-oriented software test suite evolution," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, 2019.

[7] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on software testing, verification and validation*, 2012, pp. 231–240.

[8] E. J. Rapos and J. R. Cordy, "SimEvo: A toolset for simulink test evolution & maintenance," in *2018 IEEE 11th international conference on software testing, verification and validation (ICST)*, 2018, pp. 410–415.

[9] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Automatically repairing test cases for evolving method declarations," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–5.

[10] L. S. Pinto, S. Sinha, and A. Orso, "TestEvol: A tool for analyzing test-suite evolution," in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 1303–1306. doi: 10.1109/ICSE.2013.6606703.

[11] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: a framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, San Jose, CA, USA: ACM Press, 2014, pp. 93–104. doi: 10.1145/2610384.2610419.

[12] C. Marsavina, D. Romano, and A. 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*, Victoria, BC, Canada: IEEE, Sep. 2014, pp. 195–204. doi: 10.1109/SCAM.2014.28.

[13] S. Levin and A. Yehudai, "The Co-evolution of Test Maintenance and Code Maintenance through the Lens of Fine-Grained Semantic Changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai: IEEE, Sep. 2017, pp. 35–46. doi: 10.1109/ICSME.2017.9.

[14] A. G. Yalçın and T. Tuğlular, "Studying the Co-Evolution of Source Code and Acceptance Tests," *International Journal of Software Engineering and Knowledge Engineering*, 2023.

[15] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," presented at the Proceedings of the 4th International workshop on principles of Software Evolution, 2001, pp. 37–42.

Contribution of individual authors to the creation of a scientific article (ghostwriting policy)

- Tugkan Tuglular came out with the idea of the paper and the proposed method. The proposed method has been improved by both authors.
- Ali Gorkem Yalcin has implemented the proposed visualization method and collected the data.

Sources of funding for research presented in a scientific article or scientific article itself

No funding was received for conducting this study.

Conflicts of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.