AnaCov - A novel method for enhancing coverage analysis

Mustafa Naeem, Ahmed Tahoon, Omar Ragi, Reem El-Adawi
mustafa.naeem@siemens.com ahmed.tahoon@siemens.com
omar.ragi@siemens.com reem.eladawi@siemens.com

Abstract

One of the many methods used for bug detection in software testing is the tracking and analysis of source code coverage, which helps reveal the percentage of original code covered by the software QA engineer's regression suite. This percentage can help identify any gaps that may exist in testing.

We introduce AnaCov, a tool that facilitates this process of code coverage testing. Using coverage data obtained from GCOV and stored in Git repositories, AnaCov maps testcases to functions and source files in an SQL database. This mapping enables quality assurance personnel to track coverage across time using versioned reports, and more importantly, to quickly test coverage of newly added code. This functionality is invaluable to both developers and QA engineers as it helps ensure better coverage over time, while minimizing time and disk space usage. It also allows users to merge several runs of coverage, creating a combined coverage report that is easy to analyze.

The AnaCov tool is designed for ease of use, with a user-friendly single command line interface that provides access to all functionalities, with no suite or product restrictions. Invoking the AnaCov tool allows users to map testcases to source files or functions within the original code, which enhances efficiency when analyzing coverage related to a selected subset of files.

The ability to easily query which testcases cover certain source files has a great benefit when it comes to quickly investigating coverage of code newly added to the source files. This knowledge enables better selection of pre-check-in testcases by targeting cases with the largest coverage of functions and source files under test, which reduces the time and disk space overhead necessary for adding and testing new code. Together, these functions result in higher quality code shipped to customers.

Biography

Mustafa Naeem *is a SW Team Lead QA Engineer for Siemens Digital Industries Software. He has a B.Sc. degree in Electronics and Communications from Alexandria University, Alexandria, Egypt.*

Ahmed Tahoon *is a Senior QA Engineer for Siemens Digital Industries Software. He has a B.Sc. degree in Electronics and Communications from Menofia University, Menofia, Egypt.*

Omar Mohammed Ragi *is a SW QA manager for Siemens Digital Industries Software. He has a B.Sc. degree in Computer Science from Arab Academy for Science, Technology & Maritime Transport, Cairo, Egypt.*

Reem El-Adawi *is a Director of Quality for Siemens Digital Industries Software. She holds a B.Sc., M.Sc., and Ph.D. from Ain Shams University, Electronics and Communication department, Egypt.*

1 Introduction

In today's fast-paced and technology-driven world, software developers and quality assurance (QA) engineers are under immense pressure to accelerate the pace of their software development life cycles. Meeting tight deadlines while ensuring good code quality becomes paramount in this competitive landscape. Code quality is the ultimate goal for every developer and QA engineer, as it directly impacts the reliability and performance of the software.

In the realm of computer science, code coverage emerges as a critical metric for evaluating the effectiveness of testing efforts. It quantifies the degree to which the source code of a program is executed when a specific test suite is run. Code coverage can be expressed as a percentage or a ratio, indicating the proportion of covered lines, branches, statements, or functions within the codebase.

High code coverage indicates that a significant portion of the source code has been executed during testing, suggesting a lower likelihood of undiscovered software bugs. On the contrary, low code coverage implies incomplete or missing tests, leaving room for hidden bugs or gaps in the application's functionality.

A general guideline is to aim for at least 80% code coverage, though the specific target may vary depending on the project's complexity and requirements. Achieving high code coverage yields numerous benefits, including increased confidence and trust in the codebase, reduced risk of introducing bugs or defects, saved time and effort in writing, testing, and debugging, and avoidance of rework or technical debt.

Automatically measuring code coverage at critical points in the development workflow allows developers and QA engineers to proactively identify areas with insufficient testing and take corrective actions. By integrating code coverage analysis into the development process, teams can continually monitor and improve their testing strategies, ensuring better code quality and a more robust end product.

With this goal in mind, the AnaCov tool is introduced to provide QA engineers with capabilities to ease the code coverage analysis.

2 AnaCov Philosophy

Coverage run result directories can quickly consume substantial disk space, leading to a need for frequent deletions to maintain sufficient storage capacity. When frequently adding new testcases to a regression suite in an attempt to increase code coverage, QA engineers face the challenge of preserving old coverage run results. These results must be preserved to merge them with the newly created testcases and generate a comprehensive coverage report for analysis. However, the large disk space occupied by coverage runs makes it impractical to retain them for an extended period, hindering the completion of the incremental coverage measurement process.

Additionally, the absence of a straightforward method to determine which testcase(s) cover specific source files or functions poses a challenge. Consequently, modifications to a source file require the entire regression suite to be run. With constant additions of new features and bug fixes, the QA team creates automated testcases to validate the changes. However, this process raises critical questions about the sufficiency of new testcases in covering all possible scenarios and identifying unreachable or dead code. Hence, tracking the coverage of newly added source code, functions, and lines is essential to address these concerns effectively.

The AnaCov tool built in the Go programming language [1] to tackle these challenges by providing a solution to compress and save coverage data without affecting the ability to generate coverage reports, map testcases to source files and functions, and measure incremental coverage while using publicly available applications (GCOVand LCOV) data [2,3]. In the following subsections, we define the main principles of AnaCov and show how they are incorporated into its design.

2.1. Resources optimization

The AnaCov tool prioritizes resource optimization as its most important principle. Conducting coverage runs can be time-consuming, often taking 20 times longer than normal runs. When dealing with hundreds of thousands of testcases, performing regular full coverage runs becomes challenging, as resources are finite and shared across all teams. Moreover, the large number of testcases can lead to disk space issues, making it difficult to retain coverage runs for extended analysis periods.

In scenarios with a high rate of code changes where only a few tests might be impacted, limited hardware resources may prevent running the entire regression suite for each code change. Instead, it is preferable to minimize the number of tests to be run by selecting those that cover the modified code only. This approach benefits not only QA engineers but also developers, as running fewer tests before committing code changes significantly improves turnaround time.

By optimizing resource usage and selectively running relevant tests, AnaCov empowers development and testing teams to efficiently manage their testing efforts, leading to faster testing cycles, reduced resource consumption, and improved productivity.

2.2. Usability and user experience

AnaCov's second principle revolves around usability, aiming to enhance the code coverage analysis experience for new or inexperienced QA engineers. The tool is designed to provide a mix of necessary functionalities in a single command with minimal options, enabling quick and efficient execution. By merging certain functionalities, AnaCov ensures speedy coverage analysis and swift access to results.

2.3. Centralization

Centralization is a vital concept for successful and maintainable testing automation. By centralizing key components of testing scripts and tools, development time and costs are minimized, avoiding redundant work. Maintenance becomes simpler as modifications or fixes can be applied in one location, benefiting all testcases that use the centralized tool. AnaCov embraces centralization to enhance maintainability. With multiple teams using the same coverage analysis tool, generating combined coverage reports for products sharing functionalities becomes effortless. The tool's shared database allows querying testcases executing specific functionalities across different products, streamlining testing expansion. Centralization in AnaCov optimizes efficiency, lowers costs, and fosters collaboration among teams, contributing to successful testing automation.

2.4. Modularity

AnaCov is designed with a modular approach, where each part of the tool is developed as separate code source files. This modularity ensures that if a defect occurs in one module, it does not impact other modules or pieces of code. Additionally, the modular structure simplifies the process of identifying and resolving errors, enhancing the overall robustness and maintainability of the tool.

3 AnaCov Components

AnaCov, a comprehensive code coverage tool, is built upon three primary modules that collectively facilitate efficient data management and analysis. These modules are the Data Storage module, the Search/Query module, and the Historical Coverage module (Fig. 1).

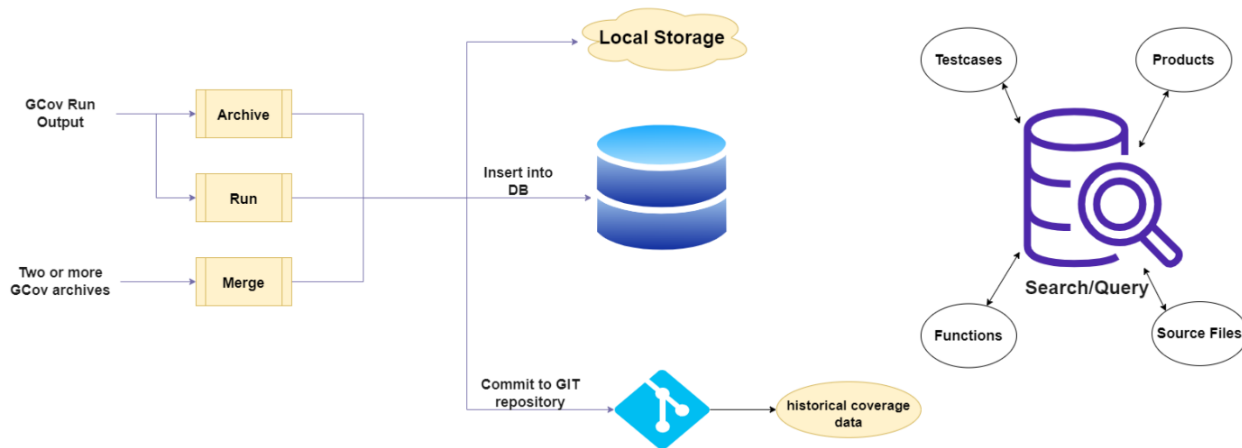

Fig. 1. AnaCov components

3.1. Modules

Regarding input/output, AnaCov takes as input the GCOV run data, which contains information about the code coverage achieved during testing. The output of AnaCov includes a detailed coverage report, showcasing the percentage of code covered by tests. Additionally, AnaCov provides a selected list of testcases, functions, or source files, enabling developers and QA engineers to focus on specific areas of the codebase that require further testing or improvement.

3.1.1. Data Storage

The Data Storage module serves as the backbone of AnaCov's functionality. It comprises two essential components: Local Storage and a Database. Local storage acts as a centralized local space in a shared location to house all the coverage data. This centralized approach ensures easy accessibility and management of data across different projects and team members. This space doesn't have any special control, it is just used to have all the data managed in one place rather than different locations for the same team members.

The database component plays a critical role in AnaCov's effectiveness. It stores the coverage output data generated during code analysis. By leveraging database capabilities, users can efficiently retrieve and utilize coverage data instantly or at a later stage, enabling them to make informed decisions and track code coverage progress. This database is supposed to keep the latest data, so for the first time a new testcase, a new source file, or a new function is added, new records are added corresponding to that in the database, otherwise the current records are updated to match the latest updates. The QA engineer has also the capability to store the data of different releases by adding a corresponding tag to the product name.

3.1.2. Search/Query

The Search/Query module acts as the user-facing interface, bridging the gap between the database and the end-users. Through this module, developers and QA engineers gain access to powerful tools that allow them to interact with and extract relevant information from the stored data. This module empowers users to perform targeted searches, obtain specific code coverage metrics, and gain insights into the quality of their codebase.

3.1.3. Historical Coverage

The Historical Coverage module is another essential component of AnaCov. It utilizes the Git version control system to maintain a comprehensive history of coverage status across different development

milestones or releases. This feature aids in tracking code coverage trends over time, facilitating a deeper understanding of codebase evolution and providing valuable insights into testing and development progress.

The key concept is that essential data for creating a coverage report is kept within an individual text file for each testcase (coverage.info). Thus, preserving this file along with its historical modifications ensures the capability to access a coverage report snapshot at any given moment (e.g., a particular release or date).

3.2. Processes

AnaCov includes several essential processes that contribute to its effective code coverage analysis.

3.2.1. Archive

The Archive process is designed to optimize disk space usage. As coverage runs can consume significant disk space, this option automatically extracts only the necessary files from the directory and generates an archive. The required files within the archive are those that store the coverage information and are essential for generating the coverage report. By archiving and compressing the necessary data, AnaCov ensures efficient storage while retaining the vital information needed for analysis.

3.2.2. Run

The Run process is the core of AnaCov functionality. It takes either the previously generated archive or the coverage data directly to generate the comprehensive coverage report. The process involves analyzing the codebase and the corresponding test suite to determine the extent of code coverage achieved. The generated coverage report provides valuable insights into the effectiveness of the testing efforts and highlights areas of the code that may require additional testing. Additionally, the Run process can feed the coverage data into the database and/or commit it to the Git remote repository, facilitating further analysis and collaboration among team members.

3.2.3. Merge

The Merge process enables the combination of multiple archives. This feature is particularly useful when dealing with large-scale projects that produce separate coverage reports for different components or modules. By merging these individual archives, AnaCov can provide a unified and comprehensive coverage analysis for the entire project. Merge ensures a holistic view of the code coverage, helping QA engineers, and developers to understand the overall test coverage across the entire codebase.

4 Archiving

When dealing with tens of thousands of testcases per product and adding more testcases on daily basis, testcase data needs tens of terabytes of disk storage. Teams should always be looking for ways to minimize the disk space that files occupy on a hard drive and reduces the time needed to transfer them. This reduction of space and time can result in significant cost savings. Compressed files require significantly less storage capacity than uncompressed files, meaning a significant decrease in storage expenses. Compressed files also require less time for transfer while consuming less network bandwidth. Which can reduce costs and increase productivity.

AnaCov provides two modes to deal with full coverage runs and archived runs.

4.1. Full Runs

In this mode, AnaCov handles full coverage runs. Since a coverage report can be generated using only coverage files, only coverage files are extracted from the coverage runs results directories and

compressed. All other testcase-related files are disposable. The compressed archives are by default added to a central directory where all product archives are kept.

In the *Archive* command, only the full unarchived coverage run argument is mandatory for this mode. There are two optional arguments: updating the DB with the coverage data, and/or updating the Git repository.

$$Anacov\ \ archive\ \ -m\ <full\,coverage\,run>\ -update_db\ \ -update_git$$

Using the *Archive* command for a Siemens product reduced the disk space of a full coverage run from 670 GB to only 20 GB. The *Archive* command deleted all the testcase-related files and kept only the coverage files, then compressed them.

4.2. Archived Runs

When interacting with a generated archive, the initial inclination may be to let the user unzip the archives manually and interact with the data this way. While this approach isn't necessarily wrong, the AnaCov application provides a more controlled environment for interacting with the archive.

The *Run* command provides an interface for interacting with the archives. It should be used to generate coverage reports from existing archives (zip files) previously generated by the *Archive* command.

This command saves the user the hassle of unzipping the archive, generating a coverage report, and cleaning up by performing all these tasks automatically.

The archived coverage run argument is mandatory for the *Run* command. There are two optional arguments: updating the DB with the coverage data, and/or updating the Git repository.

$$Anacov\ \ run\ -m\ <archived\,run>\ -update_db\ \ -update_git$$

4.3. Merge

The AnaCov *Merge* command allows user to merge multiple archives together When users have multiple zipped files from multiple products or different suites in the same product, they may need to merge them together as a method of cleanup, or for experimentation purposes.

$$Anacov\ \ merge\ output_archive\ [input_{archive1}\ input_{archive2}\ \ldots\ldots..]$$

5 Mapping

The mapping of testcases to source files and functions is a crucial and fundamental feature in the AnaCov solution. This functionality holds significant importance due to its ability to provide valuable insights into the relationships between testcases and code components, facilitating comprehensive code coverage analysis.

By establishing these mappings, QA engineers can easily determine which testcases cover specific source files or functions and vice versa. This information aids in selecting representative testcases for the functions being tested. Ensuring the proper functioning of existing functionalities is essential when adding new features to the software. Running regression tests before implementing any changes is a standard practice to validate code integrity. However, regression suites can be extensive, leading to time and resource consumption during testing. By selecting a subset of regression testcases that may be affected by the new code, QA engineers can save time and hardware resources while ensuring adequate code coverage.

For example, if a developer is modifying an existing file or function, running only the testcases that cover that file or function can be sufficient to verify that the new code does not disrupt any previously functioning functionality.

AnaCov's mapping functionality enables the generation of lists of testcases that represent specific functions or source files based on various parameters like product name, a list of source files, etc. To achieve this mapping, data must be stored and easily accessible over time. Therefore, AnaCov uses a PostgreSQL database with dedicated tables to handle testcase names, products, source files, functions, and other related information.

By employing a robust database and implementing the mapping functionality, AnaCov empowers software development teams to conduct efficient code coverage analysis, make informed testing decisions, and ensure the reliability and stability of their software products.

Fig. 2 is the diagram of the entity relationship diagram (ERD) database providing an overview of the tables, and the relations between them.

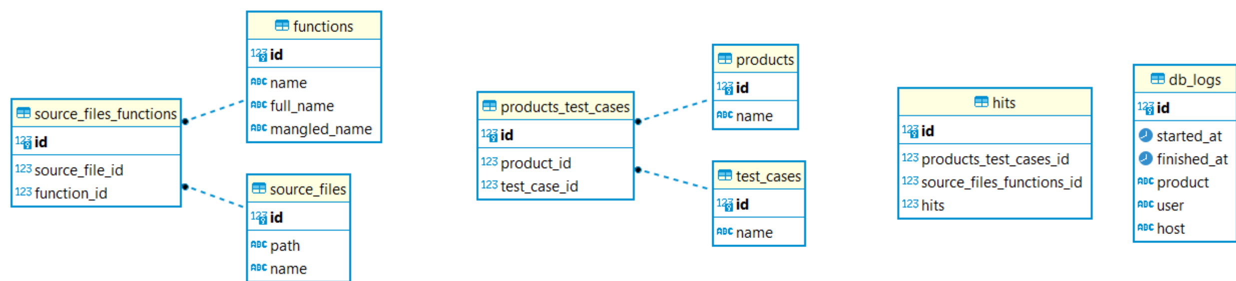

Fig. 2. AnaCov database diagram

5.1. Insertion

Data insertion into the database occurs during archiving or running processes in AnaCov. The commands provide *update_db* option to update the database with new or updated data as needed.

5.2. Querying

AnaCov offers multiple query options to identify coverage gaps and obtain relevant information. QA engineers can use queries to find testcases that cover specific files, functions, or the entire testcases for a particular product. Additionally, queries can be made to identify source files generating the covered code for specific testcases and products. The following code examples demonstrate various query options.

Testcases matching query:

$$Anacov\ \ search\ testcases\ \{-p\ [\dots products], -s\ [\dots source_files], -f\ [\dots functions]\}\ [flags]$$

Source files matching query:

$$Anacov\ \ search\ sourcefiles\ \{-t\ [\dots test_cases], -p\ [\dots products], -f\ [\dots functions]\}\ [flags]$$

Functions matching query:

$$Anacov\ \ search\ functions\ \{-t\ [\dots test_cases], -s\ [\dots source_files], -p\ [\dots products]\}\ [flags]$$

Products matching query:

$$Anacov\ \ search\ products\ \{-t\ [\dots test_cases], -s\ [\dots source_files], -f\ [\dots functions]\}\ [flags]$$

The optional *flags* provide additional functionality to the commands, such as generating output in a comma-separated value (CSV) file, limiting result numbers, or using regular expressions for searching instead of exact names for testcases, source files, or functions.

6 Incremental Coverage

As part of a new feature's validation process, QA engineers create testcases. A critical question is how to make sure created testcases efficiently cover the newly added or modified codebase of the feature under test. AnaCov's Incremental Coverage feature helps by checking the coverage of the newly added or modified code using the created testcases apart from measuring the whole coverage, which is more efficient, faster, and more optimized.

Also, whenever QA engineers need to measure coverage of a feature developed in a certain period, they must contact the developer who implemented that feature asking for a specific list of functions and source files that cover the feature only to be able to measure its coverage alone. By using the AnaCov Incremental Coverage feature, that request process is not necessary anymore.

The main idea behind generating a coverage report for the newly added or modified code only, is manipulating the instrumented lines in the cumulative coverage file. Instrumented lines are the lines that represent code lines (comments, includes, and empty lines are not instrumented lines). If the code line number is mentioned in the coverage file, it's an instrumented line that represents an executable line of code.

When QA engineers provide AnaCov with two dates of certain commits and as a future enhancement AnaCov will support an input of two source code branches, AnaCov checks which lines of code were modified or added between the two dates, and then keeps those lines only in the cumulative coverage file. Only those lines retained in the cumulative coverage file are instrumented lines and old code or unchanged code is considered to be source code comments, and its coverage is not measured.

After that, when a HTML coverage report is generated, only recently modified, or added code will be measured for coverage.

6.1. Use Model

The expected use model for AnaCov's Incremental Coverage feature involves QA engineers measuring feature coverage before starting validation and testcase creation, and then measuring it again after completing all testcases in the test plan.

Using a single command and a new coverage run for the created testcases, it becomes effortless to measure incremental coverage whenever a new testcase is created to validate a missing scenario.

Moreover, the incremental coverage mode can be employed in automated regular coverage runs, generating continuous HTML coverage reports. These reports can be used by developers and QA engineers to monitor and enhance the coverage of newly modified or added code.

The process is simplified for QA engineers, who only need new coverage run results, two dates representing the coverage report's time frame, and a list of source files to generate the report for the newly modified or added code. This seamless process facilitates efficient coverage analysis and contributes to the overall success of feature testing and code validation.

QA engineers should ensure the new date is set at most to the date the coverage run is given to the command itself or before.

$$Anacov\ \ datedcov\ -m\ < coverage\ run >\ -N\ < New\ date >\ -O\ < Old\ date >\ -f\ < source\ files >$$

Copies may not be made or distributed for commercial use

6.2. Output files and HTML report

Three main files are generated when using the *DatedCov* command:

- Dated_Coverage.info → Coverage.info file used by LCOV to generate the HTML report
- HTML_report → HTML report to visualize the functions and lines coverage
- Source_files_versions → source files versions that were checked between the two dates

The generated HTML report has three types of lines (Fig. 3):

1- Lines highlighted in blue are the lines that were executed by the testcases.
2- Lines highlighted in red are the lines that were not executed by the testcases.
3- Unhighlighted lines are comments, includes, empty lines, and lines of code that were not changed during the two dates provided by the QA engineer.

```
3420 :          if(!validation_status)
     :          {
     :              extern Pex_message_handler Global_Message_handler;
   0 :              Global_Message_handler.print_summary();
   0 :              safe_exit( 1 );
```

Fig. 3. AnaCov HTML report

7 Coverage History Tracking

The Historical Coverage feature in AnaCov utilizes the powerful Git version control system, to maintain a comprehensive record of code coverage status across different milestones or releases. When AnaCov is run over a regression suite or a part of it, the coverage data is committed to the remote Git repository, preserving a historical log of the coverage metrics.

By leveraging Git's version control capabilities, AnaCov's Historical Coverage feature empowers software development teams to gain valuable insights into code coverage trends over time. It facilitates a deeper understanding of the evolution of test coverage and aids in making informed decisions to improve code quality and testing strategies. QA engineers can use this historical data to identify patterns, assess the effectiveness of testing efforts, and ensure continuous improvement in software development processes.

7.1. Features

To retrieve and make use of this historical coverage data, AnaCov provides three essential commands: *Get*, *Peek*, and *Log*.

7.1.1. Get

The *Get* command allows users to download coverage data for a specific regression suite or a list of testcases based on various criteria. Users can specify a date or a commit hash or request the latest data (HEAD) to obtain the desired coverage metrics. For example, QA engineers can check the coverage for a specific product release or track the coverage progress from one release to another.

Examples:

A specific date: $Anacov\ \ get\ \ -product\ \ < product\ name >\ \ -date < date >$
A specific commit: $Anacov\ \ get\ testcase\ \ -product\ \ < product\ name > \ -commit < commit\ hash >$
The latest: $Anacov\ \ get\ testcase\ \ -testcases < list\ of\ testcases > \ -product\ \ < product\ name >$

7.1.2.Peek

The *Peek* command provides a list of testcase names available in a particular commit, date, or the latest data. This command is particularly useful when new testcases are added over time to cover new product features or enhance coverage for existing functionalities. By tracking these changes, QA engineers can gauge the impact on overall code coverage.

Examples:

A specific date: $Anacov\ \ peek\ \ -product\ \ < product\ name > \ -date\ \ < date >$
A specific commit: $Anacov\ \ peek\ \ -product\ \ < product\ name > \ -commit\ \ < commit\ hash >$
The latest: $Anacov\ \ peek\ \ -product\ \ < product\ name >$

7.1.3.Log

The *Log* command prints the commit logs and their associated hashes from the local repository. These commit hashes can then be used with the *Get* and *Peek* commands to access the relevant coverage data for specific commits or time ranges.

Examples:

All: $Anacov\ \ git\ log$
Date range: $Anacov\ \ git\ log\ \ -since\ \ < date > \ -until\ \ < date >$
From specific commit: $Anacov\ \ git\ log\ \ \ -from\ \ < commit\ hash >$

8 Use Models

This section will present illustrative examples of utilizing the AnaCov tool across various scenarios and demonstrating the combined use of its features to fulfill specific use cases.

When QA engineer starts the validation of a new feature, a critical aspect is assessing the extent to which newly added code will be covered by forthcoming testcases. AnaCov's Incremental Coverage feature can address this by gauging the coverage of feature-related code before and after testcase creation. This two-step coverage measurement ensures thorough coverage of the newly developed feature.

Leveraging the Git feature in AnaCov eliminates the need to retain old coverage files, enabling effortless generation of HTML coverage reports for historical coverage runs. This simplifies the presentation of coverage statistics for different source code branches.

For customized builds or daily code changes, optimizing regression testing becomes imperative. AnaCov's database facilitates the generation of a tailored list of testcases covering modified code only. This allows for targeted test runs instead of full regressions with every code alteration.

In large product environments, distinct QA engineers often test different product features. Utilizing AnaCov's *Archive*, *Merge*, and *Run* commands, coverage runs for different features in the same product can be organized and merged to reduce disk space. This consolidated archive can then be employed to update Git repo, map testcases to code elements, or generate comprehensive HTML reports for the product under test.

Enhancing partially covered source files and functions often involves modifying testcases executing code from these components. By utilizing AnaCov's database to map testcases to these files/functions, QA engineers can strategically modify testcases to boost coverage.

These are just a few illustrative scenarios of AnaCov tool application. The tool's versatile features can be combined in numerous ways to cater to diverse use models and enhance testing practices across different development contexts.

9 Conclusion

By measuring code coverage at critical points in the development workflow, developers and QA engineers can proactively identify and correct any deficiencies. Introducing automated code coverage analysis into the development process enables teams to continually monitor and improve testing strategies, ensuring better code quality and a more robust end product.

The innovative AnaCov tool provides developers and QA engineers with multiple automated capabilities to simplify, speed up, and standardize code coverage analysis. Through Its use in production environments, we demonstrate the AnaCov successfully simplifies the activities related to code coverage analysis, making the tool invaluable for software testing purposes. Users with any level of expertise can quickly and easily interface with coverage data, perform the necessary analysis, and develop a future plan to provide better code coverage in their test suites, without sacrificing huge amounts of time or disk space.

10 Acknowledgements

The authors would like to thank Siemens Calibre QA team for their help with AnaCov testing, including their insightful and helpful ideas and feedback that contributed to making the tool more mature. We would also like to thank Shelly Stalnaker for editorial assistance in the preparation of this manuscript.

References

[1] Go programming language: https://go.dev/doc/

[2] GCOV—a Test Coverage Program: https://gcc.gnu.org/onlinedocs/gcc/GCOV.html

[3] LCOV: https://LCOV.readthedocs.io/en/latest/