UDC 004.415:004.051
DOI https://doi.org/10.32782/2663-5941/2026.1.2/19

Komleva O.O.
https://orcid.org/0009-0002-2796-6449
Odesa Polytechnic National University

METHOD FOR AUTOMATIC TEST SUITE REDUCTION WITHOUT LOSS OF DETECTED DEFECTS

The paper proposes a method for automatic test suite reduction in regression testing of evolving software systems, which is based on preserving the defect detection capability of tests. Unlike conventional approaches focused on code coverage metrics or heuristic criteria, the proposed method treats the preservation of the set of detected defects as a hard constraint of the reduction process.

The proposed test suite reduction method is implemented as a sequence of interrelated steps, starting with the construction of a formalized mapping between tests and detected defects based on test execution results. This is followed by the identification of mandatory tests, a greedy selection of tests to cover the remaining defects, and a local optimization of the resulting test set with mandatory verification of defect detection preservation.

To validate the effectiveness of the proposed approach, a software prototype of an automatic test suite reduction tool was implemented. The prototype performs offline processing of automated testing results without interfering with the test execution process. It takes structured test execution data as input, including test identifiers, sets of detected defects, and test execution times, based on which the "test–defect" correspondence model is automatically constructed and stepwise test suite reduction is performed in accordance with the proposed algorithm.

An experimental study was conducted on three open-source software projects, where the size of the original test suites ranged from 320 to 740 tests and the number of reference defects ranged from 85 to 190. The experimental results demonstrated that the proposed method achieves a reduction of test suites by 41.3–57.8% and a decrease in total test execution time by 52.4–67.0% while fully preserving defect detection capability. A comparison with random and coverage-based test suite reduction approaches showed the superiority of the proposed method, as the defect preservation metric for baseline methods decreased to 0.82–0.91. The obtained results confirm the effectiveness of defect-oriented test suite reduction in modern automated regression testing processes.

Keywords: *automated testing; test suite reduction; software quality; test process optimization; defect analysis.*

Formulation of the problem. With the increasing complexity of software systems and the intensity of their evolution, the volume of automated test suites used to ensure software correctness and stability also grows significantly. In modern development processes, particularly under continuous integration and delivery conditions, an excessive number of tests leads to longer change verification times, increased computational costs, and reduced maintainability of the testing infrastructure. Under such conditions, the problem of optimizing test suites without reducing testing quality becomes especially relevant.

Common approaches to test suite reduction are usually based on code coverage analysis or heuristic identification of duplicate test scenarios. However, such methods do not guarantee the preservation of the test suite's ability to detect defects, since structural or coverage diversity of tests does not always correlate with their actual defect detection effectiveness. As a result, test reduction may lead to the loss of important scenarios, which becomes apparent only at later stages of software system operation.

In this context, an approach oriented not toward formal test characteristics but toward their actual contribution to defect detection becomes particularly important. Establishing a correspondence between tests and detected defects enables the development of reduction methods that preserve the defect detection properties of the original test suite with respect to a given reference set. Such an approach makes it possible to combine a reduction in test suite size with controlled testing quality.

Analysis of recent research and publications. The problem of test suite reduction is one of the key issues in modern regression testing, as it is directly related to reducing test execution time and computational resource consumption without compromising defect detection quality [1]. The relevance of this problem increases in continuous integration and delivery environments, where test suites evolve together with software systems.

A significant body of research has been devoted to the empirical analysis of test suite reduction in real projects. In [2], it is demonstrated that aggressive test suite reduction during software evolution can lead to a loss of defect detection capability, even when formal coverage criteria are preserved. Similarly, a comparative analysis of adequate and inadequate reduction strategies conducted in [3] showed a strong dependence of the results on the choice of test preservation criteria. These findings emphasize the need to focus not only on structural metrics but also on the defect detection properties of tests.

Systematization of existing approaches has been addressed in several survey studies. In [4], a systematic review of search-based test suite reduction methods is presented, their classification is proposed, and limitations related to scalability and result stability are outlined. The study in [5] examines the evolution of optimization practices for automated and manual testing in industrial settings, which makes it possible to consider test suite reduction as a component of the broader problem of test process optimization.

A separate group consists of methods based on heuristic and optimization algorithms. In [6], greedy test suite reduction strategies are empirically investigated, showing that they can achieve significant test reduction but do not always guarantee preservation of defect detection capability. The trade-off between the degree of reduction and testing quality is analyzed in detail in [7], where an approach to balancing these opposing objectives of reduction is proposed.

With the development of data analysis methods and artificial intelligence, approaches based on machine learning have emerged. In [8], the application of unsupervised learning methods for test suite reduction is investigated, enabling the automatic identification of redundant tests without explicitly defined selection rules. Similar ideas of scalable optimization are further developed in [9, 10], where black-box test minimization methods based on test code similarity and language models are proposed.

An important research direction is the use of information from software repositories. In [11], the possibility of reducing the number of tests by leveraging models built on historical repository data is demonstrated. The study in [12] focuses on the phenomenon of test deletion in Java applications and shows that test suite reduction in development practice often occurs informally, without explicit guarantees of testing quality preservation.

Existing approaches to test suite reduction can be conventionally divided into several main groups. Coverage-based methods focus on preserving structural code coverage but do not guarantee the preservation of the defect detection capability of tests. History-based approaches use information about defects detected in previous versions of the software; however, their effectiveness depends on the completeness of historical data. Mutation-based methods rely on artificially generated defects as a reference for testing quality and provide a closer relationship with the defect detection properties of tests, but they are characterized by high computational cost. An analysis of these approaches shows that none of them, in their basic form, combines effective test reduction with a guaranteed preservation of the set of defects detected by the original test suite.

A comparison of test suite reduction with other testing optimization strategies, such as test selection and test prioritization, was conducted in [13], showing that different approaches have fundamentally different implications for large-scale systems. In the context of modern CI/CD processes, infrastructural solutions have attracted attention, in particular the SCM-driven testing and benchmarking approach presented in [14], which provides a basis for integrating test suite reduction into industrial development pipelines.

Some studies focus on specialized application domains. In particular, [15] proposes a test suite reduction method for smart contracts, which confirms the universality of the test reduction problem and the possibility of adapting it to specific classes of software systems.

Overall, the analysis of the literature indicates that existing test suite reduction approaches provide different trade-offs between reduction effectiveness and the preservation of testing quality. At the same time, most methods do not guarantee complete preservation of the set of defects detected by the original test suite or require significant computational resources. This determines the relevance of further research aimed at developing automatic test suite reduction methods that are directly oriented toward the defect detection properties of test suites.

Task statement. The objective of this study is to develop a method for automatic test suite reduction that forms a minimized subset of tests while fully pre-

serving the set of defects detected by the original test suite. The proposed method is oriented toward practical use in automated testing processes and allows for experimental validation on real software projects.

Outline of the main material of the study. Let a software system be given for which a complete automated test suite $T=\{t_1, t_2, \ldots, t_n\}$ has been constructed. The execution of each test t_i is characterized by a finite execution time and may lead to the detection of one or more software defects. The set of defects that can be detected during testing is denoted as $D=\{d_1,d_2,\ldots,d_m\}$.

For each test, a set of defects that it is capable of detecting is defined. Accordingly, a mapping $detect:T\to 2^D$ is introduced, where $detect(t_i)\subseteq D$ denotes the subset of defects detected by test t_i. In practice, the set D is formed on the basis of historical defects of the software system or the results of mutation testing, which allows it to be treated as a reference for the defect detection capability of the test suite.

For an arbitrary subset of tests $T'\subseteq T$, the set of defects detected by this subset is defined as follows:

$$K(T') = \bigcup_{t\in T'} detect(t).$$

Similarly, $K(T)$ characterizes the defect detection capability of the complete test suite.

The task of automatic test suite reduction consists in finding a subset $T'\subseteq T$ that satisfies the following conditions:

– **preservation of defect detection capability**: $K(T')=K(T)$, that is, the reduced test suite detects all defects from the reference set D that were detected by the original test suite (i.e., $K(T)$);

– **cost minimization**: the subset T' is minimal in cardinality or minimizes the total test execution time.

Thus, the test suite reduction problem is formalized as a constrained optimization problem in which the condition $K(T')=K(T)$ is treated as a hard constraint. Since the set covering problem is NP-hard, a heuristic approach is adopted in this work that combines greedy selection with local optimization, providing an acceptable trade-off between reduction quality and computational complexity.

The proposed test suite reduction model is based on representing the relationship between tests and defects in the form of a binary matrix.

$$M = \left|m_{ij}\right|, m_{ij} = \begin{cases} 1, if\ test\ t_i\ detects\ defect\ d_j, \\ 0, otherwise. \end{cases}$$

Each row of the matrix corresponds to an individual test, and each column corresponds to a defect from the set D.

Within this model, the test suite reduction problem is reduced to a set covering problem, in which it is necessary to select the minimum number of rows of the matrix M that ensure coverage of all columns covered by the original test suite T (i.e., the set $K(T)$). To account for practical aspects of testing, weighting coefficients reflecting test execution time or cost may be introduced into the model.

The proposed problem formulation guarantees preservation of the defect detection capability of the test suite only with respect to a given reference defect set D (accordingly, preservation is ensured for $K(T)$ derived from D). At the same time, this model provides a formalized basis for constructing algorithms for automatic test suite reduction with controlled quality, making it suitable for practical use in automated testing processes.

The automatic test suite reduction method is based on the analysis of the defect detection properties of tests and the construction of a minimized subset of tests that preserves the set of defects detected by the original suite. The method consists of five sequential steps, each aimed at the gradual reduction of the test suite while maintaining control over the quality of the reduction.

Step 1. Construction of the "test–defect" mapping. At the first stage, the mapping $detect:T\to 2^D$ is constructed, which determines which defects from the reference set D are detected by each test $t\in T$. This mapping is built based on the results of test execution in a controlled environment. Historical defects of the software system or defects generated through mutation testing may be used as sources of reference defects. The result of this step is a binary "test–defect" correspondence matrix, which serves as the input data for subsequent test suite reduction.

Step 2. Identification of mandatory tests. At the second stage, tests are identified without which it is impossible to preserve the defect detection capability of the original test suite. A test is considered mandatory if it is the only test that detects a certain defect from the set $K(T)$. Formally, a test t_i is included in the set of mandatory tests T_{req} if there exists a defect $d_j\in K(T)$ such that $d_j\in detect(t_i)$ and $d_j\notin$detect(t_k) for all $t_k\in T\setminus\{t_i\}$. All mandatory tests are automatically included in the reduced test suite, and the corresponding defects are considered covered.

Step 3. Greedy test selection. After identifying the mandatory tests, a greedy selection of additional tests from the set $T\setminus T_{req}$ is performed to cover the remaining defects. At each iteration, the test that detects the largest number of yet-uncovered defects is selected.

To account for practical aspects of testing, a

weighted utility function may be used that combines the defect detection capability of a test with its execution cost:

$$score(t) = \frac{|detect(t) \cap D_{uncovered}|}{cost(t)},$$

where $D_{uncovered} \subseteq K(T)$ is the set of defects not covered at the current iteration, and $cost(t)$ denotes the execution time or another cost characteristic of the test. The procedure is repeated until all defects from the set $K(T)$ are fully covered.

Step 4. Local optimization of the result. After forming the initial reduced test set, a local optimization stage is applied, aimed at further decreasing the number of tests. At this stage, each test is temporarily removed from the set and it is verified whether the condition $K(T')=K(T)$ is preserved. If the removal of a test does not lead to the loss of any elements from the set $K(T)$ (i.e., defects detected by the original test suite), the test is permanently removed from the reduced set. This step makes it possible to eliminate redundant tests that may have been included at the previous stage due to the greedy nature of the algorithm.

Step 5. Verification of reduction correctness. The final stage consists in verifying the correctness of the obtained reduced test suite. For this purpose, the sets of defects detected by the original and the reduced test suites are compared. The reduction is considered correct if the equality $K(T')=K(T)$ holds. If this condition is violated, the reduction is considered incorrect, indicating inconsistency in the input data or in the construction of the mapping *detect*. If the defect preservation requirement is satisfied, the reduced test suite can be integrated into the automated testing process as a replacement for the full test suite.

To verify the feasibility of the proposed method, a software prototype was developed for automated test suite reduction based on the defect detection properties of tests. The implementation of the prototype is oriented toward integration with existing automated testing processes and does not require modification of the source code of the software system.

The test suite reduction prototype is implemented as a standalone tool that takes test execution results as input and produces a reduced test suite as output. The implementation is based on the Python programming language and is realized as a separate console module that performs offline processing of testing results. Input data are provided in the form of structured files, in which the following information is recorded for each test: a unique test identifier, the set of defects detected during its execution, and the test execution time. To ensure compatibility with different testing environments, tabular data representations in CSV format are supported (for the "test–defect" matrix and execution metadata), as well as export/import capabilities in JSON format for integration with pipelines and artifact storage systems.

The architecture of the prototype consists of the following main components:

– a data collection module responsible for importing test execution results, including information about detected defects and test execution time;

– a model construction module that forms the detect mapping and the corresponding "test–defect" matrix;

– a test reduction module that implements the proposed test suite reduction algorithm;

– a validation module intended to verify the correctness of the reduction and to control the preservation of defect detection capability;

– a reporting module that generates reduction results in a format convenient for analysis.

Such a component-based decomposition ensures the extensibility of the prototype and enables its adaptation to different testing environments.

The input data of the prototype are provided in the form of structured files that describe test execution results. For each test, the following information is recorded: the test identifier; the set of defects detected during test execution; and the test execution time or another cost-related characteristic. Based on these data, the detect mapping and the corresponding test–defect correspondence matrix are automatically constructed.

The reduction algorithm is implemented in accordance with the steps described in the previous section. At the stage of identifying mandatory tests, the uniqueness of defect coverage is analyzed. Next, a greedy test selection strategy is applied, with the possibility of taking into account weighting coefficients corresponding to execution time costs. The final stage involves local optimization and verification of the correctness of the reduction. To ensure reproducibility of results, all intermediate data and algorithmic decisions are stored in the internal structures of the prototype and can be reanalyzed if necessary.

The output of the prototype is a reduced test suite presented as a list of test identifiers. In addition, a report is generated that includes the following information: the number of tests before and after reduction; the total execution time of the full and reduced test suites; confirmation of the preservation of defect detection capability; and a list of defects covered by each test in the reduced suite. The obtained results

can be used for further analysis of the method's effectiveness and for integrating the prototype into automated testing pipelines.

Figure 1 presents the prototype interface of the test suite reduction tool, which supports loading input data, configuring reduction parameters, and running the algorithm in an automated mode. The interface also provides visualization of key reduction results, including the number of selected tests, the reduction ratio, execution time savings, and the defect detection preservation metric.

The implementation of the prototype is oriented toward offline analysis of test execution results and does not require direct intervention in the test execution process. At the same time, the quality of the reduction directly depends on the completeness and representativeness of the reference defect set. In the event of changes in the program code or the emergence of new defects, the prototype requires re-execution with updated input data.

The experiments were conducted on three open-source software projects that have automated unit tests and available defect data. For each project, a complete test suite T and a reference defect set D were formed based on the results of mutation testing. The size of the test suites and the number of defects varied depending on the project, which made it possible to evaluate the stability of the method under different conditions.

The objective of the experimental study was to quantitatively assess the effectiveness of the proposed method for automatic test suite reduction and to verify the preservation of its defect detection capability. The evaluation was performed by comparing the characteristics of the full and reduced test suites using a set of formalized metrics.

The following metrics were used to analyze the results:

– test suite reduction ratio

$$R = \left(1 - \frac{|T'|}{|T|}\right) * 100\%;$$

– test execution time savings

$$TS = \left(1 - \frac{\sum_{t \in T'} time(t)}{\sum_{t \in T} time(t)}\right) * 100\%;$$

– preservation of defect detection capability

$$DP = \frac{|K(T')|}{|K(T)|}.$$

Table 1 presents the aggregated experimental results for each of the studied projects.

The results of the experimental study summarized in Table 1 demonstrate the effectiveness of the proposed method for automatic test suite reduction across all considered software projects. The initial test suites contained from 320 to 740 tests, whereas

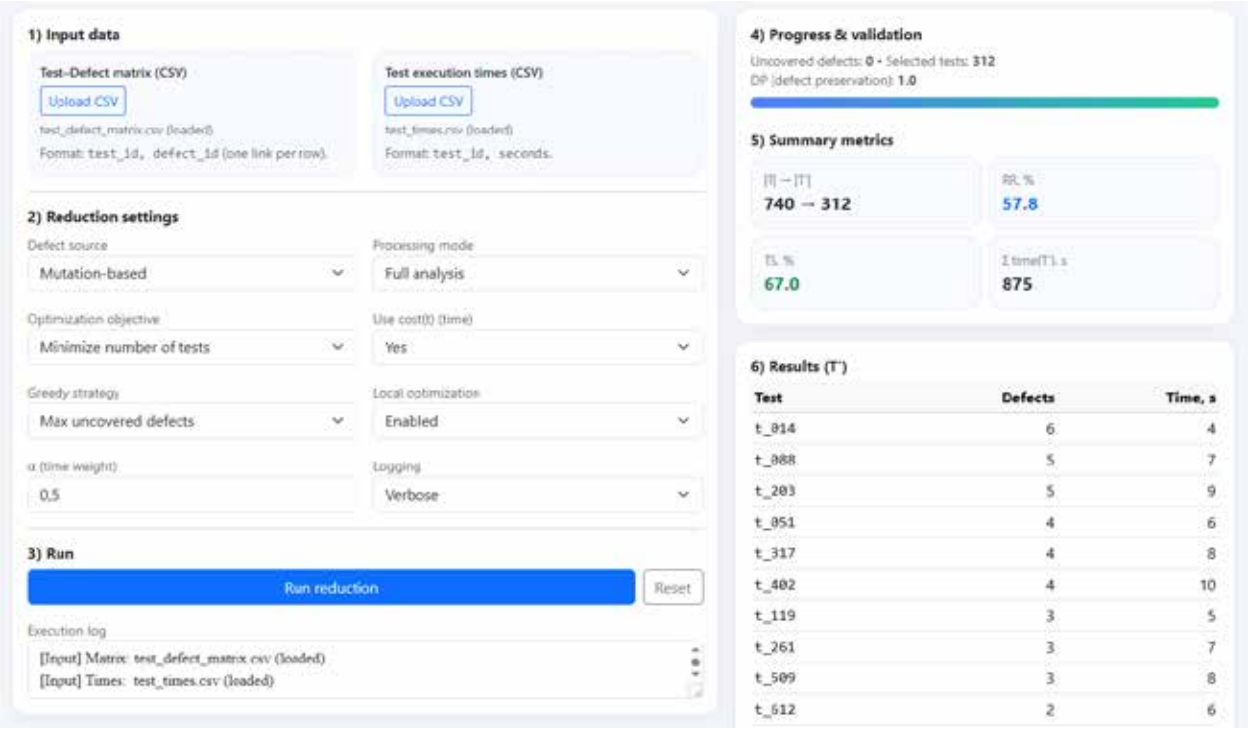

Fig. 1. Prototype interface for defect-preserving test suite reduction and reduction results

Table 1

Test suite reduction results

Metric	P_1	P_2	P_3
Number of tests (***T***)	**320**	**540**	**740**
Number of tests (***T′***)	**188**	**271**	**312**
***R*, %**	**41.3**	**49.8**	**57.8**
Execution time ***T*, sec**	**1240**	**1980**	**2650**
Execution time ***T′*, sec**	**590**	**810**	**875**
***TS*, %**	**52.4**	**59.1**	**67.0**
Number of defects (***D***)	**85**	**132**	**190**
DP	**1.0**	**1.0**	**1.0**

after applying the method their size was reduced to 188–312 tests depending on the project. Thus, the test reduction ratio increased from 41.3% for project P_1 to 57.8% for project P_3.

The analysis of time characteristics showed that the reduction in the number of tests was accompanied by an even more significant decrease in the total test execution time. For project P_1, the test execution time was reduced from 1240 s to 590 s, corresponding to a time saving of 52.4%. In project P_2, the total execution time decreased from 1980 s to 810 s (59.1%), while for project P_3 it was reduced from 2650 s to 875 s, providing the maximum time saving of 67.0%.

At the same time, the defect detection capability of the test suites was fully preserved for all projects. The number of defects in the reference set was 85, 132, and 190 for projects P_1, P_2, and P_3, respectively, and in all cases the reduced test suites detected the complete set of defects, as confirmed by the defect preservation metric *DP* equal to 1.0.

For comparison, an analysis was conducted using baseline approaches, including random test reduction and coverage-based reduction. In these cases, defect detection preservation was not guaranteed, and the value of the *DP* metric decreased to 0.82–0.91 depending on the project.

The obtained results indicate that the proposed method makes it possible to significantly reduce the size and execution time of test suites without loss of defect detection quality, which is particularly important for practical application in automated testing processes.

Conclusions. The article addresses the problem of automatic test suite reduction in regression testing with a focus on preserving the defect detection capabilities of tests. An analysis of modern approaches has shown that most existing test reduction methods are based on structural or heuristic criteria and do not provide guarantees of fully preserving the set of detected defects.

A formalized model of test suite reduction is proposed, based on representing the correspondence between tests and defects in the form of a matrix, as well as a method for automatic test suite reduction that combines the identification of mandatory tests, greedy selection, and local optimization under the condition of strict preservation of defect detection capability. To validate the proposed method, a software prototype was implemented, oriented toward integration with existing automated testing processes.

The experimental results confirmed the effectiveness of the proposed approach, demonstrating a significant reduction in the number of tests and their execution time while fully preserving defect detection capability. Further research should be directed toward accounting for defect evolution and integrating the method into CI/CD processes.

Bibliography:

1. Cruciani E., Miranda B., Verdecchia R., Bertolino A. Scalable Approaches for Test Suite Reduction. Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE 2019). Montreal, QC, Canada, 2019. P. 419–429. DOI: https://doi.org/10.1109/ICSE.2019.00055.

2. Shi A., Gyori A., Marinov D. Evaluating Test-Suite Reduction in Real Software Evolution. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’18). New York : ACM, 2018. P. 236–246. DOI: https://doi.org/10.1145/3213846.3213875.

3. Coviello C., et al. Adequate vs. Inadequate Test Suite Reduction Approaches. Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM ’18). 2018. P. 1–10. DOI: https://doi.org/10.1145/3239235.3240497.

4. Habib A. S., Khan S. U. R., Felix E. A. A Systematic Review on Search-Based Test Suite Reduction: State-of-the-Art, Taxonomy, and Future Directions. *IET Software.* 2023. Vol. 17, No. 2. P. 93–136. DOI: https://doi.org/10.1049/sfw2.12104.

5. Haas R., Nömmer R., Juergens E., Apel S. Optimization of Automated and Manual Software Tests in Industrial Practice: A Survey and Historical Analysis. *IEEE Transactions on Software Engineering*. 2024. Vol. 50, No. 8. P. 2005–2020. DOI: https://doi.org/10.1109/TSE.2024.3418191.

6. Wang Z., Li X., Chen Y. Empirically Evaluating Greedy-Based Test Suite Reduction Methods. *Science of Computer Programming*. 2017. Vol. 150. P. 1–25. DOI: https://doi.org/10.1016/j.scico.2017.05.004.

7. Shi A., Gyori A., Gligoric M., Zaytsev A., Marinov D. Balancing Trade-offs in Test-Suite Reduction. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). New York : ACM, 2014. P. 246–256. DOI: https://doi.org/10.1145/2635868.2635921.

8. Sebastian A., Naseem H., Catal C. Unsupervised Machine Learning Approaches for Test Suite Reduction. *Applied Artificial Intelligence*. 2024. Vol. 38, No. 1. Art. e2322336. DOI: https://doi.org/10.1080/08839514.2024.2322336.

9. Pan R., Ghaleb T. A., Briand L. ATM: Black-Box Test Case Minimization Based on Test Code Similarity and Evolutionary Search. Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '23). 2023. P. 1–12. DOI: https://doi.org/10.48550/arXiv.2210.16269. (date of access: 20.01.2026).

10. Pan R., Ghaleb T. A., Briand L. LTM: Scalable and Black-Box Similarity-Based Test Suite Minimization Based on Language Models. Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '23). 2023. P. 1–12. DOI: https://doi.org/10.48550/arXiv.2304.01397. (date of access: 20.01.2026).

11. Gharachorlu G., Sumner N. Leveraging Models to Reduce Test Cases in Software Repositories. Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021). Madrid, Spain, 2021. P. 230–241. DOI: https://doi.org/10.1109/MSR52588.2021.00035.

12. Bhatta S., Kendemah F., Jha A. K. Understanding Test Deletion in Java Applications. Proceedings of the 22nd IEEE/ACM International Conference on Mining Software Repositories (MSR 2025). Ottawa, ON, Canada, 2025. P. 408–420. DOI: https://doi.org/10.1109/MSR66628.2025.00071.

13. Fallahzadeh E., Rigby P. C., Adams B. Contrasting Test Selection, Prioritization, and Batch Testing at Scale. *Empirical Software Engineering*. 2025. Vol. 30. Art. 33. DOI: https://doi.org/10.1007/s10664-024-10589-8.

14. Thome J., Johnson J., Dawson I., Bolkensteyn D., Henriksen M., Art M. SourceWarp: A Scalable, SCM-Driven Testing and Benchmarking Approach to Support Data-Driven and Agile Decision Making for CI/CD Tools and DevOps Platforms. Proceedings of the IEEE/ACM International Conference on Automation of Software Test (AST 2023). Melbourne, Australia, 2023. P. 68–78. DOI: https://doi.org/10.1109/AST58925.2023.00011.

15. Górski T. Pattern-Based Test Suite Reduction Method for Smart Contracts. *Applied Sciences*. 2025. Vol. 15, No. 2. Art. 620. P. 1–18. DOI: https://doi.org/10.3390/app15020620.

Комлева О.О. МЕТОД АВТОМАТИЧНОГО СКОРОЧЕННЯ ТЕСТІВ БЕЗ ВТРАТИ ВИЯВЛЮВАНИХ ДЕФЕКТІВ

У статті запропоновано метод автоматичного скорочення тестових наборів у регресійному тестуванні програмних систем, що ґрунтується на збереженні дефект-виявлювальної здатності тестів. На відміну від поширених підходів, орієнтованих на показники покриття або евристичні критерії, запропонований метод розглядає збереження множини виявлюваних дефектів як жорстку умову редукції.

Запропонований метод автоматичного скорочення тестового набору реалізується як послідовність взаємопов'язаних кроків, що починаються з побудови формалізованого відображення між тестами та виявлюваними дефектами на основі результатів виконання тестів. Далі здійснюється виділення обов'язкових тестів, жадібний відбір тестів для покриття решти дефектів і локальна оптимізація отриманого набору з обов'язковою перевіркою збереження дефект-виявлювальної здатності.

Для перевірки працездатності запропонованого підходу реалізовано програмний прототип інструмента автоматичного скорочення тестових наборів, який здійснює офлайн-обробку результатів автоматизованого тестування без втручання у процес виконання тестів. Прототип приймає на вході структуровані дані про результати тестування, зокрема ідентифікатори тестів, множини виявлюваних дефектів і часові витрати на виконання тестів, на основі яких автоматично формується модель відповідності «тест–дефект» і виконується поетапна редукція тестового набору відповідно до запропонованого алгоритму.

Проведено експериментальне дослідження на трьох програмних проєктах з відкритим вихідним кодом, де розмір початкових тестових наборів становив від 320 до 740 тестів, а кількість еталонних дефектів – від 85 до 190. Результати експериментів показали, що запропонований метод забезпечує скорочення тестових наборів на 41,3–57,8 % та зменшення сумарного часу виконання тестів на 52,4–67,0 % за повного збереження дефект-виявлювальної здатності. Порівняння з випадковою редукцією та редукцією тестових наборів на основі покриття засвідчило перевагу запропонованого

підходу, оскільки для базових методів значення показника збереження дефектів знижувалося до 0,82–0,91. Отримані результати підтверджують доцільність використання дефект-орієнтованої редукції тестів у сучасних процесах автоматизованого регресійного тестування.

Ключові слова: *автоматизоване тестування, скорочення тестових наборів, якість програмного забезпечення, оптимізація процесів тестування, аналіз дефектів.*

Дата першого надходження статті до видання: 26.01.2026
Дата прийняття статті до друку після рецензування: 04.03.2026
Дата публікації (оприлюднення) статті: 08.04.2026