The European Journal of Research and Development

journals.orclever.com/ejrnd

OR CLEVER Science & Research Group

Research Article

Change Impact Analysis Case Study for Aviation: Mutation Testing

Nurbanu Hınık[1*,2], Özcan Çağırıcı[1], Ufuk Sakarya[3]

[1] Turkish Aerospace Industries, İstanbul, Turkey, ORCID: 0000-0001-5658-7291, nurbanuhinik1@gmail.com.tr, ORCID: 0000-0003-2215-3684, ozcan.cagirici@gmail.com

[2] YILDIZ Technical University, Graduate School of Science and Engineering, Department of Avionics Engineering, İstanbul, Turkey

[3] YILDIZ Technical University, Faculty of Applied Sciences, Department of Aviation Electrics and Electronics, İstanbul, Turkey, ORCID: 0000-0002-8365-3415, usakarya@yildiz.edu.tr

[*] Correspondence: nurbanuhinik1@gmail.com

(First received March 23, 2022 and in final form June 01, 2022)

Reference: Hınık, N., Çağırıcı, Özcan, & Sakarya, U. Change Impact Analysis Case Study for Aviation: Mutation Testing. *The European Journal of Research and Development*, *2*(2), 213–223.

Abstract

As the complexity of modern software systems increases, changes in software have become crucial to the software lifecycle. For this reason, it is an important issue for software developers to analyze the changes that occur in the software and to prevent the changes from causing errors in the software. In this paper, mutation testing as software test analysis is examined. Mutation tests have been implemented on open-source Java projects. In addition, for aviation projects, emphasis is placed on performing change impact analysis processes in compliance with the certification based on DO-178C processes.

Keywords: Change Impact Analysis, Mutation Testing, Software Testing

1. Introduction

Change is inevitable throughout the software development lifecycle. Software configuration management is the process that exists to systematically manage, control, and regulate changes in documents, code, and other assets throughout its entire lifecycle. When considering the aviation industry, the software configuration management process is critical for both military and civil aviation applications as the complexity of avionics systems increases. The main purpose of the software configuration management process is to increase production while reducing errors. One of the key activities of software configuration management is change control. Along with software change control, all factors associated with a proposed change in the software are evaluated, change priorities are determined and incorrect/undesirable changes are prevented. A change in software,

if left unchecked, can affect all assets, and have catastrophic consequences. Determining the software impact of changes in avionics software development projects is also in the field of system engineering. In this context, the concept of Trade Studies (trade-off analysis) is defined as a decision-making approach used by integrated teams to make alternatives and resolve conflicts within the framework of systems engineering [8].

RTCA DO-178C, which is a globally accepted guidance document in avionic software development projects, defines objectives for each process of the software lifecycle. In this direction, the impact of changes in the software on the whole software is expressed as "change impact analysis" in the DO-178C guidance document. According to the DO-178C guidance document, changes in an application or development environment should be identified, analyzed, and re-validated [5]. It is recommended to conduct a change impact analysis of the changes that occur in the software, with a method determined in the Change Review phase [5]. Basically, change impact analysis is required for every change [15]. Although the extent to which change impact analysis should be applied for each change that occurs is described in the DO-178C guidance document, it is not specified which methods will be used.

There are other approaches on this subject. It has been shown that graph measurements can be used to predict error-prone versions of software, with a graph method that examines the structure and evolution of software [3]. The release management process, which is an important part of the software change control process, has been modeled as seven stages, and the activities to be implemented in these stages and the roles of individuals have been defined [7]. In the study where a generative model of software dependency graphs was designed (GD-GNC) and the structure of artificial graphs created with this model were compared with real software graphs, it was examined that there is a common arrangement in software dependency graphs [12]. In the study conducted to characterize the graph-based model to be used in software and systems engineering, evaluations were made on many graph metrics and sample models [18]. Automatic generation of graph models has become an important approach in various applications of software and systems engineering. In this context, a technique is proposed to derive an automatic graph model [17]. An interactive visualization tool based on call graphs has been developed to help software developers understand the software system by focusing on specific parts of software systems separately [1]. The impact of software language selection on software quality in software development has been investigated using open-source software projects [4]. An open-source tool called GraphEvo, which can automatically generate and visualize color-coded call graphs, has been developed to help software developers better understand the software and monitor the changes that occur in the software [19]. A small change in the software system can produce large local and non-local ripple effects throughout the software system. For this reason, a large-scale mature software system was examined and the change in entropy

value was observed as the software evolved [9]. A protocol based on mutation testing is presented to quantitatively evaluate the impact analysis of errors that occur as the software evolves [13]. Change impact analysis has two major challenges, scaling to the size of software systems consisting of thousands of modules, and accuracy of impact estimates. The accuracy of the impact estimation has two points: whether the predicted impacted items are actually impacted and whether all of the actually impacted items are predicted. Accordingly, learning algorithms have been studied to improve impact estimation based on call graphs [11]. Finally, open-source software projects and determined mutation operators have been studied on the estimation of effect propagation. In this study, impact estimation was analyzed based on call graphs [14]. As seen in all these studies, every change in the software should be analyzed and controlled.

In this paper, in order to emphasize the importance of change impact analysis in software systems, mutation testing was applied with the mutation operators we determined for open-source software projects. The extent to which software systems are impacted by the changes we have made has been evaluated. We would like to emphasize the importance of change impact analysis for an Avionics software development project compatible with DO-178C processes. The projects we analyzed in our study are open source and not aviation projects. What we want to show in this study is that change effect analysis can be applied using mutation testing.

The rest of the paper is planned as follows. The related works are presented in Section 2. The proposed study is introduced in Section 3. The experimental results are demonstrated in Section 4, and finally, the paper is concluded in Section 5.

2. Related Works

2.1.DO-178C

It is a guidance document prepared to be used in the software development process in the systems and equipment of the "flying" vehicles. The purpose of the DO-178C guidance document is to ensure that the software in a system or device is compatible with airworthiness requirements at the desired level. It defines the objectives of the processes used throughout the software lifecycle, the activities that must be done to achieve these objectives, and the outputs of these activities. Software lifecycle processes defined by DO-178C guidance document are defined as software planning, software development and software lifecycle support processes; software verification, software quality assurance, software configuration management, certification liaison processes. Every process is related to each other [5].

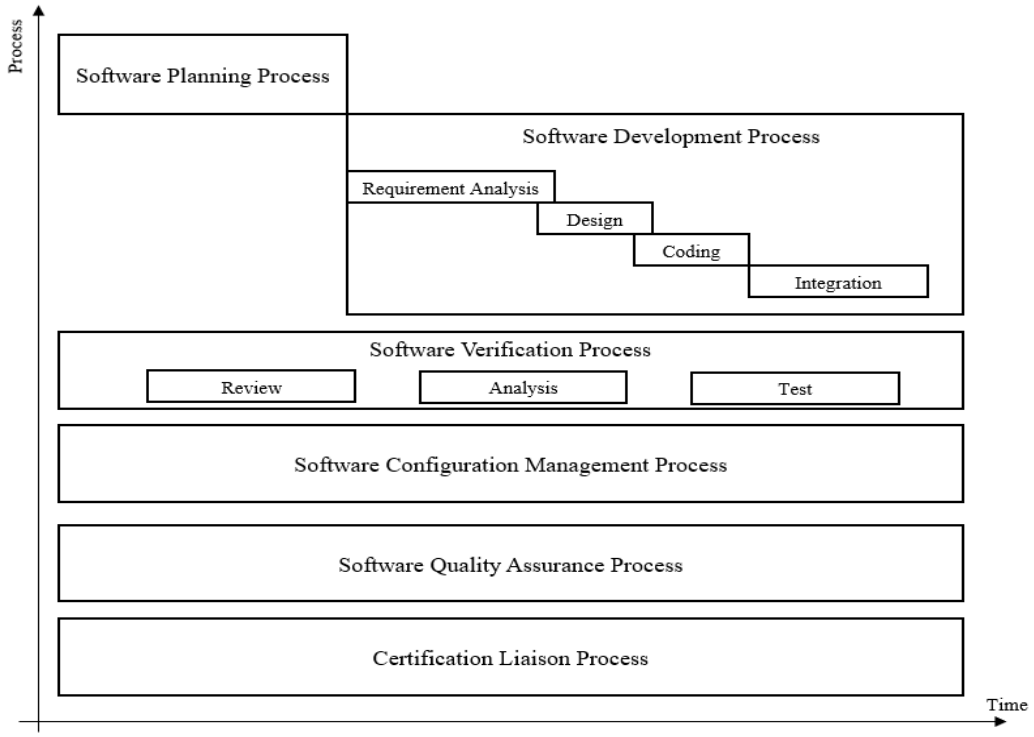

Figure -1 RTCA / DO-178C Software Life Cycle Process and Time [5]

The software configuration management process includes activities such as defining software configuration items, providing their controls, keeping configuration management records, and managing the software change process. This paper focuses on the software change management process, which is an important activity of the DO-178C guidance document's software configuration management process.

Change can occur at any process of the software development process and is a core feature of software. Changes can occur for many reasons, such as adding or changing functions, fixing a bug, performance improvements, improving processes, changing the environment. Without effective change management, chaos ensues. One of the tasks of an effective change management process is to perform change impact analysis. To properly plan the resources required to implement a change and the amount of revalidation required, the impact of the change must be analyzed [15].

2.2.Change Impact Analysis

For safety-critical systems, software changes are a way of life. In the process of designing software products, care should be taken to ensure that the software is maintainable and upgradeable. When software changes occur in safety-critical software systems, they must be carefully planned, analyzed, implemented, controlled, and verified. In a certified product, a change impact analysis is required for approved and

modified software [15]. The analyzes that should be included in a change impact analysis are described below [15][16]:

Traceability analysis: Identifies software items that may be directly or indirectly affected by the change. Traceability analysis aims at determination of the impact of the change on the software project. It is important to be able to identify both replaced and impacted software items.

Analysis of memory margins: It is carried out to ensure that the memory allocation requirements are met and protected. It cannot be completed until the change is applied. It can be predicted early, but the actual impact assessment is completed after implementation of the change.

Analysis of timing margins: It is carried out to ensure that timing requirements are met and protected. It cannot be completed until the change is applied.

Analysis of data flow: Identifies the negative effects caused by changes in the data flow.

Analysis of control flow: Identifies adverse effects caused by changes in control flow.

Input/output analysis: Identifies the impact of software changes on the input and output requirements of the product.

Development environment and process analysis: Identifies any changes that may affect the software product.

Operational characteristics analysis: It ensures that changes that may affect system operation do not adversely affect the system.

Certification maintenance requirements analysis: In case of software change, it identifies whether a new or changed certificate needs maintenance. If each change made in the software affects a certificate maintenance requirement, it is included in the scope of change impact analysis.

Partitioning analysis: Guarantees that changes to the software do not affect the mechanisms included in the design.

The software lifecycle data impacted by the change should be documented as part of the change impact analysis. It is critical for software projects to have a well-defined change impact analysis process [5]. There are many change impact analysis methods that have been used for years in the software configuration management process.

2.3.Mutation Analysis

Mutation testing is a method for increasing the code coverage of test suites. In this approach, it is built on making numerous copies of a program, known as mutants, in which one (or more than one) minor changes are added. By running test suites on each mutation, it is possible to identify source code portions that require more testing. It is modified a portion of the code (i.e., mutate the code) in each time, a little flaw is added

that test suites should identify. If the change is not identified, it implies that test suites are not adequately covering the change [10]. If all of the test pass, the mutant is considered to be alive, which suggests that the test coverage should be improved. In the contrary instance, if at least a single test fails, the mutant is called by killed, implying that the test covers the mutated region [10].

A mutation operator specifies the type and method in which items are changed. Mutation testing is an example of what is commonly referred to as error-based testing. In other words, it entails the generation of test data with the goal of detecting particular errors or classes of errors. A program undergoes a huge number of simple modifications (mutations) one at a time. Then, test data that identifies the mutated versions from the original version must be found [20].

3. Proposed Case Study: Mutation Testing

The change impact analysis process, which is an activity of the software change management process, which plays a critical role in the software development life cycle, has been examined. One of the most successful ways to protect software systems from erroneous changes is to improve the quality of software test cases. Based on this, it is proposed to apply mutation testing in order to prevent erroneous changes in software systems. Traditional test coverage only measures what code is run by software tests. It does not check if the tests can actually detect errors in the code that is being executed. For this reason, it is proposed that mutation test coverage percentages measured by applying mutation testing to a software with determined mutation operators can be used as an effective method to measure the durability of projects. In Section 4, mutation test coverage results are shown by applying mutation testing to three identified open-source Java projects.

4. Experimental Results

4.1.Dataset

To analyze changes that occur in software systems, it is considered that a dataset composed of three Java software packages: Google Gson, Apache Commons Codec, Apache Commons Lang. Table 1 contains the name, the version, the git commit-id, and the number of lines of code (computed using Jacoco [2]) of the software being analyzed.

Table 1 Statistics About the Project Considered in This Paper

Project Name	Version	Commit	Line of Code
Google Gson	2.3.2	fcfd397d	17745
Apache Commons Codec	1.11	7decf21b	50648
Apache Commons Lang	3.11	75a6aac07	74166

4.2.Implementation Details

The experiments done in this paper are implemented in the Pitest tool. An open-source tool, Pitest, was used to perform mutation testing [6]. The open source Jacoco tool was used to calculate the line of code value and test coverage percentage of the projects we analyzed [2].

Mutation operators are required for our method. The mutation operator changes only one atomic element. A mutation can include any software source code elements. The mutation operators we used in this paper are listed in Table 2.

Table 2 List of Mutation Operators Considered in This Paper

Mutation Operator	Description
Conditionals Boundary	The conditionals boundary mutator substitutes the boundary counterpart for the relational operators, =,>, >=.
Constructor Calls	The constructor calls mutator switches constructor calls with null values.
Experimental Argument Propagation	The experimental mutator substitutes method calls with one of its matching type parameters.
False Returns	Changes boxed and primitive boolean return values with false.
Math	For either integer or floating-point arithmetic, the math mutator substitutes another operation for binary operations.
Return Values	The new returns mutator set has taken the place of this mutator. The return values mutator modifies method call return values. A different mutation is used depending on the method's return type.
True Returns	True is used to replace primitive and boxed boolean return values.

4.3.Results

Within the scope of the projects we analyzed, the mutation test result in line with the mutation operators we determined is given in Table 3. Table 3 contains the project name, the name of the mutation operator used in this paper, number of classes, percentage of unit test line coverage, classes in which mutation operators are available unit test coverage (Unit Test Line Coverage), percentage of mutation test line coverage, mutation test coverage of functions where mutation operators are available (Mutation Test Line Coverage), percentage of mutation test strength, number of mutation test (Test Strength) of the software being analyzed.

The comparison of unit test coverage values and mutation test coverage values on the basis of the examined projects is given in Table 4. Table 4 contains the project name, the number of lines of code (computed using Jacoco [2]) percentage of unit test coverage and percentage of mutation test of the software being analyzed. Although the result obtained from this comparison seems to have a high unit test coverage, the coverage value decreases as a result of the mutation test. This shows that the unit test coverage value can not enough to detect errors in the source code.

5. Conclusion

Any change in the software may affect the entire software system. Software testing is essential to building an effective and fault-tolerant software system. The emphasis in this paper is the use of mutation testing methods to prevent errors that may occur in the software development life cycle. In this paper, the case study we have created on open-source Java projects shows that by measuring the scope of software tests with mutation testing, we show that mistakes in change situations can be prevented. The importance of applying the change impact analysis process in certification-compliant aviation projects is emphasized in the guidance document DO-178C. Although we have shown our results in this paper through open-source Java projects, our main goal is to increase the software quality by using these methods in safety-critical aviation projects. Our future work is to create change impact analysis methods for DO-178C compatible software by using these methods in open-source aviation projects.

Table 3 Mutation Analysis Results

Project Name	Mutation Operator	Number of Classes	% Unit Test Line Coverage	Unit Test Line Coverage	% Mutation Test Line Coverage	Mutation Test Line Coverage	% Test Strength	Test Strength

Google Gson	Conditionals_Boundary	11	92	1751/1913	57	37/65	59	37/63
	Constructor_Calls	37	87	3267/3755	63	292/460	97	292/300
	Experimental_Argument_Propagation	23	89	2218/2486	54	66/123	61	66/109
	False_Returns	17	88	1492/1696	74	51/69	93	51/55
	Math	13	91	1781/1959	70	164/235	72	164/228
	Return_Vals	41	88	3463/3953	76	559/734	92	559/605
	True_Returns	18	86	2000/2317	68	71/105	93	71/76
Apache Commons Codec	Conditionals_Boundary	38	95	3859/4063	67	168/251	67	168/250
	Constructor_Calls	50	94	3830/4053	81	268/329	94	268/286
	Experimental_Argument_Propagation	47	95	3970/4183	82	416/509	88	416/472
	False_Returns	23	96	1983/2070	82	45/55	88	45/51
	Math	34	96	3520/3648	92	1354/1468	95	1354/1421
	Return_Vals	56	95	4367/4616	82	696/850	92	696/754
	True_Returns	30	95	2627/2773	85	89/105	92	89/97
Apache Commons Lang	Conditionals_Boundary	60	96	12172/12689	55	735/1341	55	735/1338
	Constructor_Calls	103	96	13782/14404	82	798/974	96	798/833
	Experimental_Argument_Propagation	122	96	14153/14793	77	1196/1549	80	1196/1502
	False_Returns	91	96	12158/12717	90	528/587	97	528/542
	Math	65	97	11465/11821	82	1299/1592	83	1299/1571
	Return_Vals	177	95	15765/16551	88	4126/4692	95	4126/4353

True_Returns	90	96	11896/12438	80	559/699	89	559/627

Table 4 Comparison of Project-Based Unit Test Coverage and Mutation Test Coverage

Project Name	**Line of Code**	**Unit Test Coverage Percentage**	**Mutation Test Coverage Percentage**
Google Gson	17,745	84	69
Apache Commons Codec	50,648	97	85
Apache Commons Lang	74,166	95	81

References

[1] Alanazi, R., Gharibi, G., & Lee, Y. (2021). Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, *176*, 110945. https://doi.org/https://doi.org/10.1016/j.jss.2021.110945

[2] baeldung. (2016). Intro to JaCoCo | Baeldung. In *Baeldung*. https://www.baeldung.com/jacoco

[3] Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. *2012 34th International Conference on Software Engineering (ICSE)*, 419–429. https://doi.org/10.1109/ICSE.2012.6227173

[4] Bhattacharya, P., & Neamtiu, I. (2011). Assessing Programming Language Impact on Development and Maintenance: A Study on c and C++. *Proceedings of the 33rd International Conference on Software Engineering*, 171–180. https://doi.org/10.1145/1985793.1985817

[5] Firme, R. (2011). *Software considerations in airborne systems and equipment certification*. Rtca, Inc.

[6] *http://pitest.org*. (n.d.).

[7] Kajko-Mattsson, M., & Yulong, F. (2005). Outlining a model of a release management process. *Journal of Integrated Design & Process Science*, *9*, 13–25.

[8] Lightsey, B. (2001). *SYSTEMS ENGINEERING FUNDAMENTALS SUPPLEMENTARY TEXT*. THE DEFENSE ACQUISITION UNIVERSITY PRESS FORT BELVOIR. https://acqnotes.com/wp-content/uploads/2017/07/DAU-Systems-Engineering-Fundamentals.pdf

[9] Maisikeli, S. (2016). Evaluation of Software Degradation and Forecasting Future Development Needs in Software Evolution. *International Journal of Software Engineering & Applications*, *7*, 49–64. https://doi.org/10.5121/ijsea.2016.7604

[10] Musco, V. (2016). *Analyse de la propagation basée sur les graphes logiciels et les données synthétiques* (Issue 2016LIL30053) [Université Charles de Gaulle - Lille III]. https://tel.archives-ouvertes.fr/tel-01398903

[11] Musco, V., Carette, A., Monperrus, M., & Preux, P. (2016). A Learning Algorithm for Change Impact Prediction. *2016 IEEE/ACM 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 8–14. https://doi.org/10.1109/RAISE.2016.010

[12] Musco, V., Monperrus, M., & Preux, P. (2017). *A Generative Model of Software Dependency Graphs to Better Understand Software Evolution*.

[13] Musco, V., Monperrus, M., & Preux, P. (2015). An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis. *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, 60–64. https://doi.org/10.1109/AST.2015.20

[14] Musco, V., Monperrus, M., Preux, P., Yin, X., Musco, V., Neamtiu, I., & Roshan, U. (2019). A large-scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal*, *25*(3), 921–950. https://doi.org/10.1109/AITest.2019.000-1

[15] Rierson, L. (2013). *Developing safety-critical software : a practical guide for aviation software and DO-178c compliance*. Taylor & Francis.

[16] Rierson, L. K. (2001). Changing safety-critical software. *IEEE Aerospace and Electronic Systems Magazine*, *16*(6), 25–30. https://doi.org/10.1109/62.931137

[17] Semeráth, O., Nagy, A. S., & Varró, D. (2018). A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. *Proceedings of the 40th International Conference on Software Engineering*, 969–980. https://doi.org/10.1145/3180155.3180186

[18] Szárnyas, G., K\Hovári, Z., Salánki, Á., & Varró, D. (2016). Towards the Characterization of Realistic Models: Evaluation of Multidisciplinary Graph Metrics. *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 87–94. https://doi.org/10.1145/2976767.2976786

[19] Walunj, V., Gharibi, G., Ho, D. H., & Lee, Y. (2019). GraphEvo: Characterizing and Understanding Software Evolution using Call Graphs. *2019 IEEE International Conference on Big Data (Big Data)*, 4799–4807. https://doi.org/10.1109/BigData47090.2019.9005560

[20] Woodward, M. R. (1993). Mutation testing—its origin and evolution. *Information and Software Technology*, *35*(3), 163–169. https://doi.org/https://doi.org/10.1016/0950-5849(93)90053-6