M.A. Álvarez-García, "Automation and evaluation of mutation testing for the new C++ standards". *ICSE 2021 SRC - ACM Student Research Competition.*

Version: Accepted Version

Automation and evaluation of mutation testing for the new C++ standards

Miguel Ángel Álvarez-García
Department of Computer Science and Engineering
Universidad de Cádiz
Cádiz, Spain
miguelangel.alvarez@uca.es

***Abstract*—Mutation testing is becoming increasingly widely used to evaluate the quality of test suites, especially to test programs coded in widely used programming languages in the industry. Mutation tools have arisen to automate the technique in different languages, including C++. With the increasing use of this technique, new mutation operators modeling possible faults often emerge to improve its abilities and adapt the tools to new advanced features. In this work, mutation operators for the new C++ standards, defined in previous work, are implemented and applied to generate and execute mutants in real programs. With this study, the MuCPP mutation tool is updated with the inclusion of these new operators. In addition, the improvements suggested in the definition of those operators can be finally tested, and conclusions about their utility in practice can be drawn. The implemented operators are checked on a set of four C++ programs that use these advanced features. The results show significant differences with the previous manual analysis: the number of invalid mutants was reduced by 64%, and we found fewer alive mutants (88%) and an increase in dead mutants (31%). In summary, both the number of mutants incorrectly classified in the previous manual analysis and the number of mutants generated (particularly equivalent mutants) have been reduced.**

***Index Terms*—Software testing, mutation testing, C++ new standards.**

I. Introduction

Software testing is a fundamental component in the development of any software project to evaluate and improve the quality of the programs. The presence of faults in the software, not detected in the testing stage, can have serious consequences, especially in critical systems. This testing phase can represent more than 50% of the total cost of a project [1]. Nowadays, the importance of software testing is even more notable in the context of Industry 4.0, where enterprises are moving towards a completely connected environment and their software systems are increasingly complex. In this scenario, extensive testing activities may result in higher costs if they are carried out completely manually.

In this context, the automation of software testing can partially solve these problems, incorporating techniques that increase the degree of confidence that the system is free of faults. One of the most powerful techniques in this area is *mutation testing* [2], which consists of introducing small syntactic modifications in the source code of a particular program and observe if this change is identified, or not, with the tests implemented. These faults are injected based on predefined transformation rules, which simulate usual programming errors, and are known as *mutation operators*.

Focusing on C++, this language has experienced a significant transformation in the last decade with the entry of new standards. This notable evolution of C++ requires attention for the testing process to be effective in new programs that adopt these characteristics. With this aim, a set of mutation operators directly related to the new standards was defined in a previous work [3]. However, such operators were not implemented in a mutation tool and, therefore, they could not be put into practice. As a consequence, a comprehensive study of the real effectiveness of these new operators is still pending.

Developing mutation testing regarding the characteristics of the new standards is necessary because the new C++ projects in the industry will adopt these advanced features progressively. In this paper, we implement the previously designed operators to sort out this issue, and apply them to real recent projects for the first time, obtaining more precise results of their behavior than previous manual analysis.

II. Background and related work

Recently, the C++ standard committee agreed to issue a new standard for the language each three years. As a result, C++11 was upgraded with different features in 2014, calling it C++14 [4]. Among the most important features inserted in C++11 and C++14, we can find the reserved word *auto*, intelligent pointers, move semantics and lambda functions.

Different researches have been done around mutation testing in the last few years [5]. Many tools for different languages have been developed to apply mutation testing automatically. In the case of C++, several tools can be found [6]–[8]. In this paper, we focus on MuCPP [9], a tool developed at the University of Cadiz to automate a set of traditional and class operators (with respect to the object-oriented part). These operators were implemented according to the C++03 standard.

The evolution of C++ through the new standards and their increasing use in the industry led Parsai et al. [3] to define new mutation operators for these revised versions of C++. They presented a set of four new mutation operators that were tested manually in their work. The four mutation operators are: range-based “for” reference removal (FOR), lambda reference capture (LMB), forced rvalue forwarding (FWD) and initializer list constructor (INI).

III. Proposed solution and novelty

In this paper, we propose the extension of the MuCPP tool so that it can be used to test programs that make use of the new features of C++. To illustrate this, and for the sake of brevity, we will focus on one of those operators: the FOR operator.

A. Operator definition

The range-based reference removal operator is based on the possible confusion between value and reference semantics when using the new range-based for loop. This operator modifies the sentences with the form *for (T& elem: range)* or *for (T&& elem: range)*, in which T is *auto* or a particular type of the language. The operator removes the reference qualifier from the range statement (& or &&), as shown in Table I.

TABLE I
FOR operator mutants

Source code	Mutants
for (auto& elem: range){...}	for (auto elem: range){...}
for (auto&& elem: range){...}	for (auto elem: range){...}

The implementation of FOR is achieved by searching for this type of loops and removing the existing references in them. In addition, the implementation of this operator has been refined to reduce the generation of equivalents and useless mutants, based on the indications of the authors of the operator. This improvement consists in omitting those loops marked as *const*. In this case, mutating the variable targeted by this operator would be useless because its value could not be modified anyway inside the loop.

B. Programs

For this work, we used four of the programs (listed in Table II) indicated in the previous work, which make use of the feature that FOR focuses on. We used the test suites implemented by their developers for each of these programs.

TABLE II
Programs to be used in the experiment

Project	Commit	Code lines	Commits number
Corrade	ff3b351	6500	1898
EntityX	6389b1f	9000	296
Json	a09193e	8000	1973
Antonie	59deb0d	9000	306

- **Corrade** [11] is a C++11/14 utility library.
- **EntityX** [12] is an Entity Component System that uses C++11 features.
- **Json** [13] is a C++ library to work with JSON.
- **Antonie** [14] is a processor of DNA reads.

C. Mutant classification

In their paper, Parsai et al. [3] manually counted the number of mutants that these new operators would produce in those programs. They also reasoned about the nature of those mutants, classifying them into *dead* (when they thought that a test case could be designed to detect the mutant) or *equivalent* (when they thought there was no way to detect it). In our paper, however, we actually automate the operators, generate the mutants and execute them against a real test suite. As a result, our classification of mutants is different from the one in the previous paper:

- A dead mutant is the one detected by the tests.
- An alive mutant is the one that is not detected, including possible equivalent mutants.

IV. Results

This operator is executed using the MuCPP tool, which generates one mutant per each mutation location found. Later, each project is compiled —invalid mutants are then detected— and the tests included in each project are run —alive and dead mutants are then identified. We can observe in Table III the results obtained from the application of FOR on the aforementioned programs and the execution of their respective tests on the resulting mutants. We show both, our automated results (A) and the results of the manual analysis in the previous paper (M), for their comparison. As can be seen in all cases, the number of invalid mutants in our execution is far fewer than the number of valid ones. It seems that the improvement incorporated helps avoid the generation of some invalid mutants. The mutants of two of the four chosen projects (EntityX and Json) remained alive after the test execution, which may indicate that these projects probably need the design of particular tests to kill these type of mutants. On the contrary, the test suites for the Corrade and Antonie projects are adequate with respect to the mutants generated since all of them have been killed. It should be noted that, as can be seen, there is a notable difference between the results of the automated and the manual analysis. This gives evidence of the importance of automating mutation operators to more precisely know about their behavior in real projects.

TABLE III
Results obtained

Project	Total		Invalid		Alive		Killed	
	M	A	M	A	M	A	M	A
Corrade	24	15	1	1	13	0	10	14
EntityX	2	2	0	0	2	2	0	0
Json	1	2	0	0	0	2	1	0
Antonie	39	18	10	3	18	0	11	15

V. Contributions

This work allows covering the new C++ standards in real projects with the extension of the MuCPP mutation testing tool. This is a timely contribution because industrial software projects will soon completely embrace these advanced features and, therefore, the automation of mutation operators focused on these standards should not be longer postponed. Thanks to this, a mechanism is established to carry out large-scale studies on the usefulness of these operators.

REFERENCES

[1] G. J. Myers, T. Badgett, and C. Sandler, "The Art of Software Testing" Wiley , Hoboken , 2011.

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in Advances in Computers, vol. 112, Elsevier, 2019, pp. 275–378.

[3] A. Parsai, S. Demeyer, and S. De Busser, "C++11/14 Mutation Operators Based on Common Fault Patterns" Testing Software and Systems. Springer International Publishing, Cham, pp. 102–118, 2018.

[4] S. Meyers, "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14" O'Reilly Media, Incorporated, Sebastopol, 2014.

[5] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, (5), pp. 649-678, 2011.

[6] "Mutate++ - A C++ Mutation Test Environment" https://github.com/nlohmann/mutate_cpp (accessed Jan. 20, 2021).

[7] "Dextool Mutate" https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate (accessed Jan. 20, 2021).

[8] A. Denisov and S. Pankevich, "Mull It Over: Mutation Testing Based on LLVM," 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 2018, pp. 25-31, doi: 10.1109/ICSTW.2018.00024..

[9] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for C++ with the MuCPP mutation system", Inf. Softw. Technol., vol. 81, pp. 169–184, 2017.

[10] L. Lampropoulos, M. Hicks and B. C. Pierce, "Coverage guided, property based testing," Proceedings of ACM on Programming Languages, vol. 3, (OOPSLA), pp. 1-29, 2019.

[11] "Corrade: C++11/C++14 multiplatform utility library." https://github.com/mosra/corrade (accessed Jan. 20, 2021).

[12] "EntityX: A fast, type-safe C++ Entity-Component system." https://github.com/alecthomas/entityx (accessed Jan. 20, 2021).

[13] "JSON for Modern C++." https://github.com/nlohmann/json (accessed Jan. 20, 2021).

[14] "Antonie: an integrated, robust, reliable and fast processor of DNA reads." https://github.com/beaumontlab/antonie (accessed Jan. 20, 2021).