Adaptable Mutation Testing For Continuous Integration Environments

Ali Parsai, Alessandro Murgia, and Serge Demeyer
University of Antwerp
Middelheimlaan 1
2020 Antwerpen, Belgium
Email: {ali.parsai, alessandro.murgia, serge.demeyer}@uantwerpen.be

Coen De Roover
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene, Belgium
Email: cderoove@vub.ac.be

***Abstract*—With the introduction of agile methods, the importance of a high-quality test suite became a crucial factor in the development of fault-free software. Mutation testing is known to be able to accurately measure the quality of a test suite. However, it is computationally intensive and consequently has not yet found widespread adoption in industry. Therefore, we propose to adopt the Cha-Q change-centric meta-model to improve mutation testing and make it more efficient for continuous integration environments.**

I. Introduction

One of the major goals in software engineering is to provide developers with tools that can automate the mundane tasks, and as such, ease the software maintenance. One of those mundane tasks is software testing. This task is indispensable as long as we want to develop a software system without faults during software maintenance. The advent of agile processes with their emphasis on test-driven development [3] and continuous integration [4], [11] implies that developers want (and need) to test their changed components early and often [18]. As a result, the test suite is a crucial part of the development cycle.

After the introduction of agile development techniques, the focus on testing has shifted more on the quality of the test suite. Consequently, the metrics adopted to evaluate such quality have become important for both industry and academia. For the former, it is a practical means to improve the effectiveness of the testing practices; for the latter, it is important to assess the validity of agile methods and comparison of different testing strategies. Mutation testing [12], [9] provides a repeatable and scientific approach to measure the quality of the test suite, and it is proven to simulate the faults realistically [2], [14]. This is due to the fact that the faults introduced by each mutant are modeled after the common mistakes developers often make [13]. Mutation testing consists of two phases: first, generating faulty versions of the code by injecting a single fault *(creating a first-order mutant)*, or multiple faults *(creating a higher-order mutant)* into the code and then, executing the test suite on this faulty version of the code to determine the outcome. The result is calculated by the percentage of the faults that resulted in failure of at least one test *(killed mutants)* divided by the total number of created mutants.

Although the idea of mutation testing has been introduced in the late 1970s, it has not found widespread use in real scenarios due to its computationally intensive nature. Even though mutation testing is proven by academic research to be de jure approach to quantify test suite quality, difficulties of this technique has caused simple code coverage metrics (e.g. statement and branch coverage) to become de facto standard test suite quality metric in industry. Therefore to be practical in industrial settings, it is crucial to smoothen the integration of mutation testing in the continuous integration cycle, and make use of the benefits [22] of this technique. However, this is not an easy task for four reasons. First, the overall process of building and testing the software is often complicated. Second, the tools used during the build process are not easily replaceable. Third, there is a lack of mature mutation testing tools and as consequence major build management systems have not developed common standards or requirements for such tools. Fourth, there is a lower demand for mutation testing due to a lack of background knowledge about the advantages it offers. For this reason, the development of new mutation testing tools is hindered, and their smooth integration is unlikely. Therefore, introduction of a tool capable of easy integration is the first step in popularizing the use of mutation testing.

To overcome the problem of integration, we propose a solution based on the Cha-Q[1] infrastructure[2] [8]. The Cha-Q environment aims to strike a balance between agility and reliability through change-centric quality assurance tools. This environment offers a first-class representation of software artifact changes. Making use of this infrastructure allows easy integration with compatible continuous integration tools, and easier transformation into new settings supported by Cha-Q infrastructure.

Our proposal is (i) to create a mutation testing tool on top of Cha-Q meta model, (ii) use ChaQeko/X (a tool under development in Cha-Q). ChaQeko/X is a program transformation tool based on Ekeko/X [7]. Its main characteristic is to be template-driven. We exploit this characteristic to provide customizable mutation operators, and adaptable to the needs

[1]Change-centric Quality Assurance
[2]http://soft.vub.ac.be/chaq/

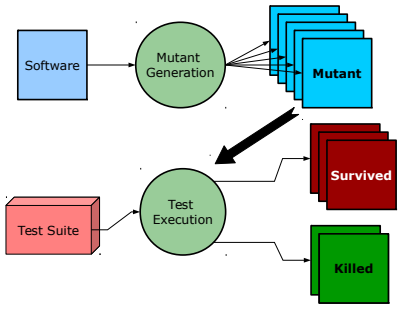

Fig. 1. Mutation testing procedure

of the developer.

II. Background

This section provides background information about mutation testing, and program transformation.

a) Mutation Testing: Mutation Testing is a fault-based testing technique to quantify the quality of a test suite. The idea of mutation testing was first mentioned in a class paper by Lipton (as reported by Offutt et al. in [21]) and later developed by DeMillo, Lipton, and Sayward [9]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [5].

The procedure for mutation testing is as follows: First, faulty versions of the software are created by introducing a single fault into the system *(Mutation)*. This is done by applying a known transformation on a certain part of the code *(Mutation Operator)*. After generating the faulty versions of the software *(Mutants)*, the test suite is executed on each one of these mutants. If there is an error or failure during the execution of the test suite, the mutant is regarded as killed *(Killed Mutant)*. On the other hand, if all tests pass, it means that the test suite could not catch the fault, and the mutant has survived *(Survived Mutant)*. An overview of this procedure can be seen in Figure 1.

$$Mutation\ Coverage = \frac{Killed\ Mutants}{All\ Mutants} \quad (1)$$

The final result is calculated using Equation 1. This metric provides a detailed image of the quality of a test suite, as it makes sure that the kind of faults simulated by the mutation operators are covered by the tests and therefore reducing the chance of missing such faults in the final product.

b) Mutation Operators: A mutation operator is a known transformation that introduces a single fault into the code. The first set of the mutation operators designed were reported in [15]. In 1996, Offutt et al [20] identified the set of sufficient mutation operators. This reduced set of operators remained more or less intact in future research papers.

With the popularity of the object-oriented programming paradigm, there was a need to design new mutation operators to simulate the bugs that occur only in this kind of programs. Several studies proposed new mutation operators [17], [6], and some demonstrated the usefulness of object-oriented operators [16], [10]. Nevertheless, sufficient mutation operators are by far the most implemented set of mutation operators, and only few tools implement the newer object-oriented mutation operators.

III. Proposed Idea

The adoption of the Cha-Q infrastructure allows us to offer three advantages over classical mutation testing. For each we briefly describe the context of the problem along with our proposed solution.

A. *Customizable mutation operators*

One important aspect of mutation testing is the use of mutation operators to generate mutants. These mutation operators are usually based on a simple fault model. However, this simplicity usually means that lots of *mutable statements* are found, and as a result lots of mutants are generated. This leads to adverse effects on performance of this procedure, and does not allow mutation testing to scale to larger software.

The use of simple mutation operators to generate the injected faults is justified based on two fundamental assumptions. First, Competent Programmer Hypothesis [9], [1] states that a competent developer will only make mistakes that can be solved by few syntactic changes. Second, Coupling Effect Hypothesis states that "complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants" [19]. Even though both these hypotheses are true for simple units, the integration of objects introduces complex new faults that cannot be detected and solved as easily. Thus, there is a need for more complex fault models to be introduced to tackle this problem. We propose to create mutation operators using code templates. Using ChaQeko/X allows us to give the developer the option to define their own mutation operators. These mutation operators can be more complex than the simple mutation operators that are being widely used. This also means that they will generate less mutants, and they can target types of faults that are relevant to the project itself rather than the generic faults.

B. *Migration to new platforms with little effort*

Generally, mutation testing tools are designed to work in a certain environment (e.g. language, platform, or continuous integration system). It is not easy to adapt mutation testing tools to work on environments other than the ones they were originally planned for. This is because the choice of mutation operators, build structures, and the parsers are environment-dependent. We propose the adoption of Cha-Q infrastructure in order to make our mutation testing tool more environment-agnostic. Since the tool will rely on Cha-Q infrastructure for most of its environment-dependent components (such as parsers, and code manipulators), once Cha-Q infrastructure is deployed, the tool can be migrated to the new platform with little effort. In Addition, all other tools built upon Cha-Q can be integrated as well. This also makes it more enticing for

companies to adopt, since they would get several useful tools at once, reducing the cost of deployment for each of them.

C. *Mutation testing in small incremental steps*

Generally, mutation testing needs to be performed once per revision. This means that even if two revisions of the same project are rather similar, still the mutation testing must be performed completely on both. We propose to use the facilities offered by the Cha-Q infrastructure to identify the changes between two revisions, and apply mutant testing only on the entities that are relevant to those changes. In this manner, we reduce the computational effort to perform mutation testing heavily, allowing it to be employed in a continuous integration environment efficiently.

IV. Conclusion

Mutation testing is a proven method of quantifying test suite quality. However, it is computationally intensive, and it has not found widespread use in industry. Thus, we propose an idea with three distinct advantages over classical mutation testing to increase the chance of its adoption by industry:

1) Customizable mutation operators
2) Migration to new platforms with little effort
3) Mutation testing in small incremental steps

Using Cha-Q infrastructure, we can build a tool on top of ChaQeko/X with the advantages described above.

Acknowledgements

This work has been sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled Change-centric Quality Assurance (CHAQ).

References

[1] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, DTIC Document, 1979.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411, May 2005.

[3] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[5] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980. AAI8025191.

[6] Huo Yan Chen and Su Hu. Two new kinds of class level mutants for object-oriented programs. In *Systems, Man and Cybernetics, 2006. SMC '06. IEEE International Conference on*, volume 3, pages 2173–2178, Oct 2006.

[7] C. De Roover and K. Inoue. The ekeko/x program transformation tool. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 53–58, Sept 2014.

[8] Coen De Roover, Christophe Scholliers, Viviane Jonckers, Javier Prez, Alessandro Murgia, and Serge Demeyer. The implementation of the cha-q meta-model: A comprehensive, change-centric software representation. *In Electronic Communications of the European Association of Software Science and Technology, Post-Proceedings of the 8th International Workshop on Software Quality and Maintainability (SQM14)*, 65, 2014.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[10] Anna Derezinska and Marcin Rudnik. Quality Evaluation of Object-oriented and Standard Mutation Operators Applied to C# Programs. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 42–57, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Martin Fowler. Continuous integration. Technical report, http://www.martinfowler.com/, May 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

[12] R.G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, SE-3(4):279–290, July 1977.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, Sept 2011.

[14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? Technical Report UW-CSE-14-02-02, University of Washington, 2014.

[15] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[16] Hyo-Jeong Lee, Yu-Seong Ma, and Yong-Rae Kwon. Empirical Evaluation of Orthogonality of Class Mutation Operators. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 512–518, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. Inter-class mutation operators for java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 352–363, 2002.

[18] John D. McGregor. Test early, test often. *Journal of Object Technology*, 6(4):7–14, May 2007. (column).

[19] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[20] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996.

[21] A.Jefferson Offutt and RolandH. Untch. Mutation 2000: Uniting the orthogonal. In W.Eric Wong, editor, *Mutation Testing for the New Century*, volume 24 of *The Springer International Series on Advances in Database Systems*, pages 34–44. Springer US, 2001.

[22] BenH. Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.