Muteria: An Extensible and Flexible Multi-Criteria Software Testing Framework

Thierry Titcheu Chekam
thierry.titcheu-chekam@uni.lu
SnT, University of Luxembourg
Luxembourg

Mike Papadakis
michail.papadakis@uni.lu
SnT, University of Luxembourg
Luxembourg

Yves Le Traon
yves.letraon@uni.lu
SnT, University of Luxembourg
Luxembourg

ABSTRACT

Program based test adequacy criteria (TAC), such as statement, branch coverage and mutation give objectives for software testing. Many techniques and tools have been developed to improve each phase of the TAC-based software testing process. Nonetheless, The engineering effort required to integrate these tools and techniques into the software testing process limits their use and creates an overhead to the users. Especially for system testing with languages like C, where test cases are not always well structured in a framework.

In response to these challenges, this paper presents *Muteria*, a TAC-based software testing framework. *Muteria* enables the integration of multiple software testing tools.

Muteria abstracts each phase of the TAC-based software testing process to provide tool drivers interfaces for the implementation of tool drivers. Tool drivers enable *Muteria* to call the corresponding tools during the testing process. An initial set of drivers for KLEE, Shadow and *SEMu* test-generation tools, *Gcov*, and *coverage.py* code coverage tools, and *Mart* mutant generation tool for C and Python programming language were implemented with an average of 345 lines of Python code. Moreover, the user configuration file required to measure code coverage and mutation score on a sample C programs, using the *Muteria* framework, consists of less than 15 configuration variables.

Users of the *Muteria* framework select, in a configuration file, the tools and TACs to measure. The *Muteria* framework uses the user configuration to run the testing process and report the outcome. Users interact with *Muteria* through its Application Programming Interface and Command Line Interface. *Muteria* can benefit to researchers as a laboratory to execute experiments, and to software practitioners.

KEYWORDS

software testing, framework, extensible, test adequacy criteria, multi-tools, multi-languages

ACM Reference Format:
Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2020. Muteria: An Extensible and Flexible Multi-Criteria Software Testing Framework. In *AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387903.3389316

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
AST '20, October 7–8, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7957-1/20/05…$15.00
https://doi.org/10.1145/3387903.3389316

1 INTRODUCTION

Test adequacy criteria (TAC) based software testing has gained more attention among researchers and is becoming widely used in practice[1, 13]. A TAC-based software testing process, as depicted in Figure 1 (more details in section 2) involves using TACs to evaluate and improve test suites. Several phases (steps) of the software testing process optimize the execution (represented with dashed lines in the Figure 1). Many tools and techniques have been developed to help software developers test their software [1, 2, 5, 10, 11, 13]. These tools and techniques are used to increase the fault detection, provide some guarantee of the software correctness and reduce the cost of software testing through automation of the process [5]. Nevertheless, with the proliferation of programming languages, TACs and software testing tools, developers need to exert supplementary effort to learn to use newly-developed tools, and integrate them into their test environment. Furthermore, researchers exert much effort to implement and evaluate their developed techniques and often, a great deal of engineering effort is required in order to integrate their implementation with other tools. These challenges are mainly affecting programming languages such as C, where, system tests are not always well structured (can be a set of bash scripts) and the data is represented by each tool regardless of the others.

This paper presents *Muteria* in response to those challenges. *Muteria* provides a collection of simplified drivers interfaces for integration of software testing tools (implementing different aspects of the TAC-based software testing process). Tools are integrated into *Muteria* through drivers that implement interface functions to enable *Muteria* to call the tools. These drivers can be made publicly available with the corresponding tools.

The *Muteria* framework provides:

- The Flexibility to add support for new TACs and programming languages.
- An Interface to implement drivers to integrate new tools.
- A controller that handles the integration of the tools.
- A Reporter that computes metrics and display results.

The remaining of the paper will present in section 2 the motivation for building *Muteria*. In section 3, an overview of *Muteria* is presented and, a case study is shown in section 4. Finally, the related works and conclusion are presented in sections 5 and 6 respectively.

2 BACKGROUND AND MOTIVATION

2.1 Software Testing Process

Figure 1 presents an overview of a test adequacy criteria (TAC) based software testing process, adapted from the "Two mutation processes" presented by Offut [5]. During the process, tests, that are

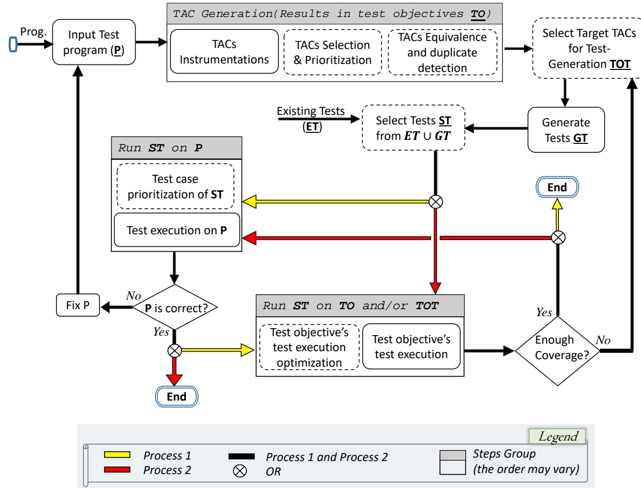

Figure 1: Test Adequacy Criteria (TAC) based software testing process (adapted from Offut's "Two mutation processes" [5]). The *Process 1* is adapted from the "Traditional process" and *Process 2* from the "Post-Mothra Process"

either manually or automatically generated, are executed (after possible selection/prioritization) on the program under test (PUT) P, to check for failures due to potential faults (in the presence of faults, the process is interrupted, the user repairs the program and restarts the process). The test suites are evaluated using TACs' coverage and improved to maximize TACs' coverage. The TACs' test objectives are generated by instrumenting the PUT. For faster execution, the TACs' test objectives of interest may be selected (for instance, a random number of mutants are selected based on mutation operators in the case of mutation testing [9] or, most likely to be faulty statements are selected in case of statement coverage). The tests are executed (with possible optimization such as optimizing strong mutation using weak mutation [4]) on the TACs' instrumented programs and the coverage values are computed. The computed TACs' coverage values are reported to the user who, based on the values, may generate more tests to increase the coverages.

There are two variants of the process: in *process 1*, the targeted TACs coverage is reached before the PUT is checked for correctness while in *process 2*, the PUT is checked for correctness before the TACs' coverage is measured.

Each phase of the preceding process have been subject to research leading to development of new techniques and tools. Nonetheless, researchers exert a great deal of engineering effort to build prototypes of their techniques which, often, are not easy to use due to the engineering effort needed in order to integrate them into the software testing process. Moreover, the experimental evaluations of the developed techniques require that scripts are implemented to integrate the prototypes with other existing tools.

2.2 Why A New Framework?

The reasons behind *Muteria* are to provide the following.

A laboratory framework for TAC-based software testing research that allows researchers to implement and evaluate their techniques with little effort.

Simplify the development of TAC-based software testing tools by providing out-of-the-box integration with other existing tools.

Ease the use of TAC-based software testing techniques through rich user interfaces and configuration.

3 *MUTERIA* FRAMEWORK OVERVIEW

We believe that a well designed software testing framework should be easy to use, provide good user interfaces and be easy to modify for different uses. *Muteria* framework implements the different phases of the TAC-based software testing process, depicted in Figure 1, with extensible interfaces. *Muteria* uses a modular approach [8] for the implementation of its functionalities.

3.1 Design Goals

In this section, we present the main features of *Muteria* that support its design.

3.1.1 Extensible. The modular design of *Muteria* framework separates the phases of the TAC-based software testing process into different components. Within each component, multiple tools that implement the corresponding phases can be integrated into the framework. Integrating a new tool into *Muteria* simply requires to extend the corresponding component's *tool driver* interface. Such a design enables the development of drivers for new tools on a specific component, independently of the tools used in other components.

3.1.2 Configurable. *Muteria* provides a wide space of configurations that allow the users to have deep control over the execution of the framework. The framework allows the users to configure the execution of the software testing process by specifying the test adequacy criteria to use during testing, whether to reuse preceding execution data or not (useful for example for regression testing), the level of concurrency and, which metrics to report and how to report them. The underlying test generation tools, test adequacy criteria tools and test execution optimization techniques can also be configured collectively or individually (tool specific configuration).

3.1.3 Multi Programming Language Support. The *Muteria* framework separately supports multiple programming language by integrating the testing tools of the same programming language. For instance, using the framework on a C language program allows only the use of tools supporting C programs. Therefore, the framework's extension tools are grouped by programming languages.

3.2 User Interaction

Muteria framework provide 2 main forms of user interaction.

3.2.1 Application Programming Interface. Users can integrate *Muteria* into other frameworks through its application programing interface (API). Moreover, *Muteria*'s components can be used as libraries to build different frameworks.

3.2.2 Command Lines. As most frameworks, *Muteria* provides a rich command lines interface (CLI), allowing users to execute the framework from terminals.

3.3 Architecture

Figure 2 presents an overview of the architecture of the *Muteria* framework. The core of the framework is made of the following components:

3.3.1 Controller. This component organizes the tasks to be executed, based on the configuration, and calls the relevant components for each of the executions. It implements the integration between the tools implementing different phases of the software testing process.

3.3.2 Code Manager. This component manages the code repository of the PUT. It also provides functions to build code (convert from one code representation to another).

3.3.3 Test Cases Manager. This component provides an abstraction of test generation and test execution to the framework. Multiple test generation and test execution tools can be integrated through drivers on this component. Manually written tests are also managed by this component. This component provides high level functions to generate and to execute tests. These functions are mapped to the underlying tools through the tool drivers.

3.3.4 TAC Manager. Similar to the Test Cases Manager, this component provides an abstraction of each implemented TAC's instrumentation tool. Multiple TACs tools can be integrated through drivers on this component. Each tool may implement support for multiple TACs. This component provides functions to instrument the PUT for the given TACs and to execute a test set against the instrumented programs (by calling the Test Case Manager).

3.3.5 Test Execution Optimizer. This component provides functions to select and prioritize tests cases (e.g. for regression testing). New test execution optimizing techniques' implementations can be integrated into this component.

3.3.6 Test Generation Guidance. This component implements functions to select, during the test generation process, "important" TACs' test objectives to focus on (e.g. select likely fault revealing statements or mutants [11] and use them to guide automated test case generation to reveal potential faults).

3.3.7 TAC Execution Optimizer. This component provides functions to select and/or prioritize TACs' test objectives for execution (e.g. using weak mutation to improve execution time of strong mutation [4]). This is useful, for instance, for strong mutation in regression testing, where the optimizer could statically select the mutants likely to be relevant to the area of interest in the program under test.

3.3.8 Reporters. This component provides functions to compute useful metrics (such as code coverage, mutants subsumption and execution time) and present to the user.

3.4 Implementation

The *Muteria* framework is implemented in Python programming language. The extension tools' drivers are also implemented in Python programming language. The integrity of the code repository of the PUT is ensured using git[1] (some TACs, e.g. mutation, may modify source files). There have been many challenges in the development of the framework, and the greatest were the modularization of the framework and design of tool driver interfaces. The *Muteria* framework is publicly available[2] open source. Installation is done by running *pip install muteria*. A docker image is also available[2].

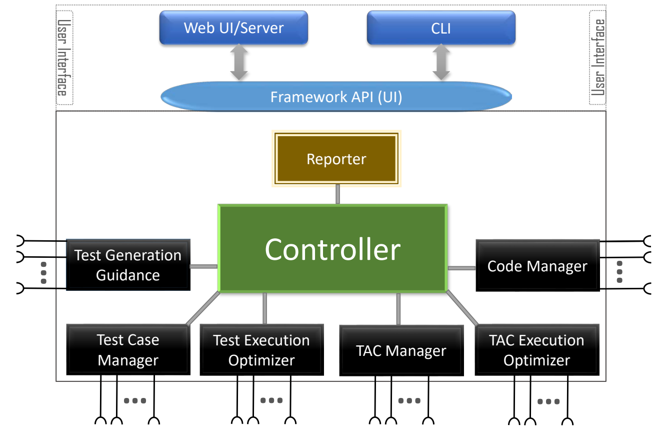

Figure 2: Architecture of *Muteria* framework. The components with black rectangle provide interfaces for corresponding tools to connect to the framework. The controller enable the integration. All components are accessible by the users through the framework API.

4 CASE STUDY

We implemented a set of drivers for several C programming language software testing tools, namely: GNU Gcov code coverage measurement tool, *Mart* [12] mutant generation tool (based on LLVM[3] and usable through command line interface), KLEE [2] test generation tool, Shadow [6] symbolic execution-based patch test generation tool, and *SEMu* [3] mutant test generation tool. We also implemented drivers for Python code coverage measurement tool *Coverage.py*[4]. Table 1 summarizes the implementation sizes of the drivers.

We also present in Table 2 the number of configuration variables that the user needs to provide to run the testing process on a software using the selected tools.

The sample reported coverage information for the execution of *Muteria* on a sample C program is shown in Figure 3.

5 RELATED WORKS

Many frameworks have been designed and developed to support TAC-based software testing. Most of those frameworks either focus on specific programing languages, specific TAC or support specific test runners. Moreover, very often, there is no straightforward approach to integrate those with other tools. Stryker [1] is an open-source mutation testing framework that supports several programming languages (currently three) and enable integration with multiple test runners. Nevertheless, currently, Stryker neither

[1] https://gitpython.readthedocs.io/en/stable/

[2] https://github.com/muteria/muteria

[3] https://llvm.org/

[4] https://github.com/nedbat/coveragepy

Table 1: *Muteria* in Practice: Implemented drivers sizes (Python LOC) for several tools

Tool Name	Tool Type	Driver Size (LOC)
GNU Gcov	Statement, Branch, Function Coverage	404
Mart	Mutant Coverage and Strong/Weak Mutation	452
KLEE	Test generation	438
Shadow	Patch Test generation	304
SEMu	Mutation Test generation	236
custom	Manually built system tests	237

Table 2: *Muteria* in Practice: Some User Configurations

Language	Testing scenario	# Config. Vars.
Python	Measure unit tests Statement/branch Coverage	8
C/C++	Measure system tests and generated tests Statement, Branch, Function Coverage and weak, strong mutation scores	16

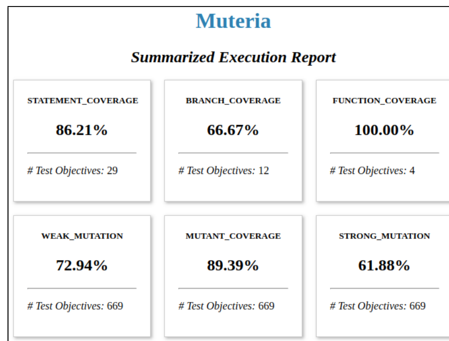

Figure 3: report of software testing with *Muteria*.

provides support for adding test adequacy criteria nor supports integration with various mutation tools or test generation tools. Open Code Coverage Framework (OCCF) [10] is a framework that aim to simplify the development of code coverage measurement in multiple programming languages. OCCF does not provide mechanisms to integrate such coverage measurement tools with other types of tools such as test generation tools. OCCF is orthogonal with *Muteria* and can be used alongside *Muteria* by developping *Muteria* drivers for the tools developed with OCCF. The Mothra mutation framework [5] was built with the goal to be expandable and adaptable. In fact, Mothra tool-set was designed to be like a *laboratory* for future research [5], which is also an important philosophy for *Muteria*. Nevertheless, Mothra was designed for Fortran programs and for mutation TAC. *Muteria* learned from Mothra and generalized to support different TACs and programming languages.

6 CONCLUSION

This paper presents *Muteria*, a framework that integrates tools developed for software testing. *Muteria* framework can be extended to supports other programming languages and provides the flexibility to add support for new test adequacy criteria (TAC). *Muteria* provide simple interfaces to implement drivers for various software testing tools such as test-case prioritization, mutant selection, test-generation, etc, tools. *Muteria* also provide rich configuration options. The main limitation of *Muteria* is the possible loss of performance due to the generalizability. In fact, some test execution optimization techniques that are language specific may not be usable with *Muteria*. Moreover, *Muteria* inherits the performance limitation of the integrated tools.

Muteria framework can be used by researchers to experiment with their findings and run experiments, and also by practitioners. *Muteria* framework was developed as a result of the challenges encountered while conducting previous research [7, 11, 13]. *Muteria* is publicly available: https://github.com/muteria/muteria.

ACKNOWLEDGMENTS

This work was supported by the AFR PhD Grant of the National Research Fund, Luxembourg, to Thierry Titcheu Chekam.

REFERENCES

[1] 2020. Stryker Mutation Framework. https://stryker-mutator.io/. Accessed: 24-01-2020.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[3] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2020. Killing Stubborn Mutants with Symbolic Execution. arXiv:cs.SE/2001.02941

[4] Sang-Woon Kim, Yu-Seung Ma, and Yong-Rae Kwon. 2013. Combining weak and strong mutation for a noninterpretive Java mutation system. *Software Testing, Verification and Reliability* 23, 8 (2013), 647–668. https://doi.org/10.1002/stvr.1480 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1480

[5] Jeff Offutt. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (2011), 1098 – 1107. https://doi.org/10.1016/j.infsof.2011.03.007 Special Section on Mutation Testing.

[6] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1181–1192. https://doi.org/10.1145/2884781.2884845

[7] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant Quality Indicators. In *the 13th International Workshop on Mutation Analysis (Mutation 2018)*.

[8] David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[9] Goran Petrovic and Marko Ivankovic. 2018. State of Mutation Testing at Google. In *40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2018, May 27 - 3 June 2018, Gothenburg, Sweden.*

[10] Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2010. Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages. In *2010 10th International Conference on Quality Software*. 262–269. https://doi.org/10.1109/QSIC.2010.42

[11] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2019. Selecting fault revealing mutants. *Empirical Software Engineering* (18 Dec 2019). https://doi.org/10.1007/s10664-019-09778-7

[12] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1080–1084. https://doi.org/10.1145/3338906.3341180

[13] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 597–608. https://doi.org/10.1109/ICSE.2017.61