Towards Mutation Testing of Simulink Models

Joanna Kisaakye[1], Onur Kilincceker[1,2] and Serge Demeyer[1,2]

[1]*Department of Computer Science, University of Antwerp, 2020 Antwerp, Belgium*
[2]*Flanders Make vzw, Belgium*

Abstract

Mutation testing offers stronger test adequacy than conventional structural coverage metrics. It employs mutants, which can mimic the real software faults, to measure the fault detection capacity of a given test suite. Mutation testing has been shown to work on an industrial scale for embedded systems, however only for unit tests. Since it is common practice to use model-based design to analyze, and test embedded software, it is worthwhile to investigate mutation testing in the context of MATLAB/Simulink. In this paper, we propose a framework for mutation testing of Simulink models. We validate the framework by means of a line follower robot constructed with Lego/Mindstorm and modelled using a series of Simulink models. This study also addresses possible open research and development challenges.

Keywords

Mutation Testing, Simulink Models, Embedded Software, Model-based Design

1. Introduction

Software testing is an indispensable activity of modern software development to measure software quality. This activity needs to be planned and executed carefully to avoid possible software related failures. Code based coverage metrics are normally used to assess adequacy of software testing methods, but only show whether a test touches the unit under test and not whether the test reveals a defect. Today's complex software systems demand not only practical and effective methods but also corresponding coverage metrics. Mutation coverage is utilized to address these concerns as being the state-of-the-art criterion for evaluating the strength of a software test suite [1].

Mutation testing employs mutation operators to obtain mutants of the software, which are faulty versions of the software and can mimic the real faults [2, 3]. Therefore, a detected fault refers to a "killed" while an undetected fault refers to a "live" mutant for corresponding software. To assess adequacy, the software testing methods evaluated with respect to mutation coverage resulted from the application of mutation testing. Mutation coverage is simply the ratio of killed mutants to total number of mutants neglecting the equivalent mutants that behave as the original (fault-free) software and are very hard to kill.

BENEVOL'21: The 20th Belgium-Netherlands Software Evolution Workshop, December 07–08, 2021, 's-Hertogenbosch (virtual), NL
Joanna.Kisaakye@student.uantwerpen.be (J. Kisaakye); onur.kilincceker@uantwerpen.be (O. Kilincceker); serge.demeyer@uantwerpen.be (S. Demeyer)
https://www.uantwerpen.be/en/staff/onur-kilincceker_23090/ (O. Kilincceker); https://win.uantwerpen.be/~sdemey/ (S. Demeyer)
0000-0001-7081-5385 (J. Kisaakye); 0000-0001-5996-4398 (O. Kilincceker); 0000-0002-4463-2945 (S. Demeyer)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3071/paper15.pdf

Mutation testing has been applied to various kinds of programming languages at the unit level, the integration level, and the specification level [4, 5]. However, the use of mutation testing for embedded software is a rather immature area of research and needs different methods than conventional mutation testing. Indeed, modern embedded software is run on specific hardware systems with respect to various constraints such as time and memory. In automotive for instance, embedded systems can grow to 100 million lines of code running on 70 to 100 microprocessors [6]. At that scale, mutation testing at the code level becomes impractical: we need to move to a higher abstraction level: model-based design. Indeed, to address the scalability issue, one promising avenue of research called model-based mutation testing employs abstract models rather than software programs [7, 8, 9, 10, 11]. The model-based mutation testing methods mainly focus on conformance relations between the abstract models and their corresponding software programs. These methods assume the presence of a correct abstract model, generate test cases from the model and use mutation testing to select the best test cases.

MATLAB/Simulink, developed by The MathWorks, enables engineers to design, analyze, and test embedded software by using model-based design. It permits testing at early-stage software artifacts on simulations that can be automatically generated from Simulink models. To this end, Simulink offers a solution to apply mutation testing on higher level design of embedded software.

This paper aims to show feasibility of mutation testing on Simulink software artifacts. It offers an automation framework to automate mutation testing to cope with testing of complex embedded software systems at earlier phases to target unit level faults. Also, we introduce a pilot study to conduct feasibility of the proposed framework. To the best of our knowledge, this is the first work to uncover mutation testing on Simulink software artifacts based on the related work provided in the next section.

The next section presents related work on mutation testing on Simulink models or its corresponding artifacts including mutation operators. Section 3 introduces the proposed approach. In section 4, we propose a pilot study to apply the proposed framework. Section 5 provides discussion on open challenges including research and development opportunities. We conclude our work in Section 6.

2. Related Work

Application of mutation testing on Simulink models or its software artifacts is an immature area and there exist several challenges based on definition of mutation operators and automatization of the processes.

Zhan and Clark proposed a method for search-based automatic test generation and measured its adequacy using mutation testing for a Simulink model [12]. They used a targeted heuristic search algorithm to capture a test that has the ability to kill mutants and compared their methods with random testing. Their work confirmed that even for a limited set of mutants (30 mutants were generated for a system of six blocks), mutation coverage can be used to assess the adequacy of a test suite.

Brillout et al. used a bounded model checking method for test data generation aimed to achieve high mutation coverage for Simulink models [13]. Their approach attempts to find a

Table 1
Comparison of Related Studies

Study	Purpose	Artefact	Cons
[12]	Test Generation	Model	Limited Mutation Operators
[13]	Test Generation	Model	Cost of Model Checking
[14]	Test Generation	Model	Manual Work
[16]	Model-clone detection	Model	Semi-Automatic
[17]	Mutation Testing	Model	-
Ours	Mutation Testing	C++ code	Missing Evaluation

counterexample for demonstrating discrepancy between original and the mutated model. The counterexamples are used to obtain test data. They plan the experimental evaluation as a future work.

Hanh and Binh presented mutation operators for mutation testing of the Simulink model to address the most common faults made by designers [14]. They divided fault classes for Simulink models into six categories and deduced five categories of mutation operators for which they informally defined different types of mutation operators. For a quadratic model pilot study, they generated 59 mutants and wrote four sets of test cases. They achieved a 0,98 mutation score. However, the mutation testing is applied manually that affects validity and reliability of their results. As follow-up research, Hanh et al. also proposed a test data generation method for mutation testing by using an artificial immune system for Simulink models [15].

Rather than mutation testing for Simulink software, Stephan et al. applied mutation testing for Simulink models for detecting model-clones and defined component level mutation operators [16]. They evaluated the mutation operators on a case study for multiple versions of three Simulink projects. They separated results for each mutation operator in terms of mutation score to measure their effectiveness. Their work was performed semi-automatically.

Besides manual and semi-automated works for mutation testing of Simulink models, Phil et al. proposed an open-source toolset called SIMULTATE to automate the entire process including injecting faults [17].

Based on the provided related work, we conclude that mutation testing of Simulink models is feasible. However, there is not yet a consensus on an effective suite of mutation operators and more experience reports are certainly welcome as well. More importantly, no study has been conducted on the application of mutation testing to Simulink software artifacts. Table 1 compares related studies considering their purpose, artifacts applied on, and their disadvantages compared with our proposal. To this end, the current work proposes an automation framework to address this shortcoming.

3. Proposed Approach

We intend to construct an automation framework for mutation testing of Simulink models integrated in state-of-the-art continuous integration (CI) and continuous delivery (CD) build pipelines. The design of the framework centers around three steps: test design, mutant genera-

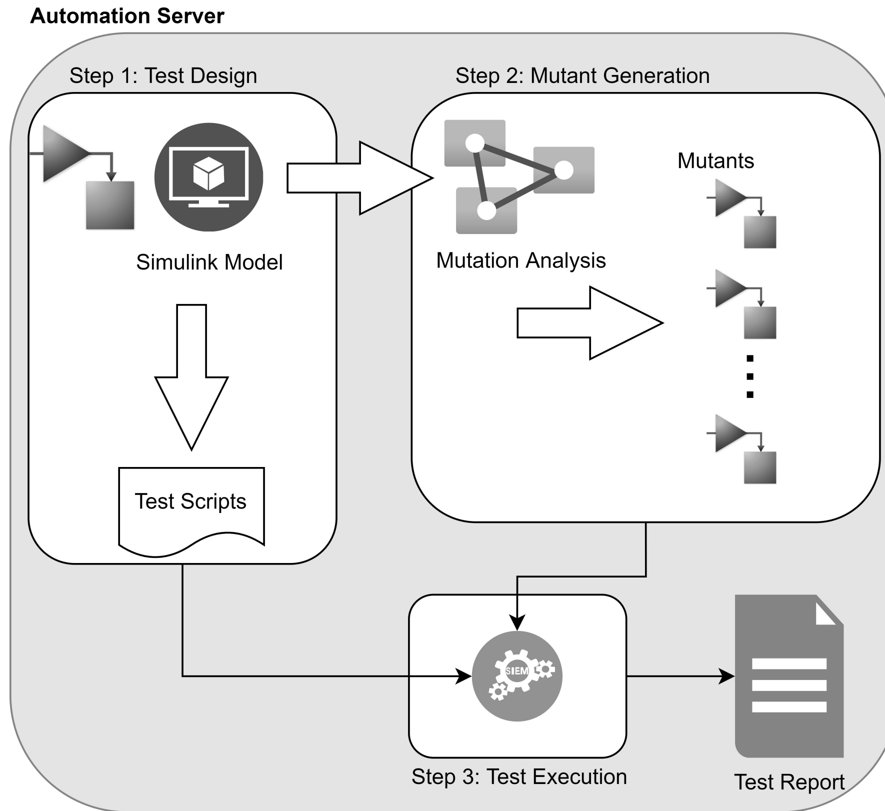

Figure 1: The proposed framework

tion, and test execution, respectively (see Figure 1).

In the test design step, unit level and model level test case will be written by using the Simulink Test environment. This step is critical for addressing specific fault types that can occur in the Simulink model. There needs to be correlation between mutants generated at the second step to these types of faults and their corresponding test cases. Therefore, this step requires expertise on designing test cases and mutation testing.

In the second step, the mutants of the Simulink model (written in the C++ programming language) will be generated by utilizing an open-source mutation analysis tool. We envision the use of Dextool to automatically generate mutants. Dextool employs specific mutation operators for the C++ programming language. These operators are ABS (Absolute Value Insertion), UOI (Unary Operator Insertion), LCR (Logical Connector Replacement), AOR (Arithmetic operator replacement), and ROR (Relational Operator Replacement). We may consider additional mutation operators, such as proposed by Parsai et al. [18] and Delgado-Pérez et al. [19].

Test scripts are executed automatically on the generated mutants in the third step. The execution process will be carried on the Simulink Test environment. Based on these executions, the framework is to output a test report that contains the overall mutation score and specific mutation scores for mutation operators.

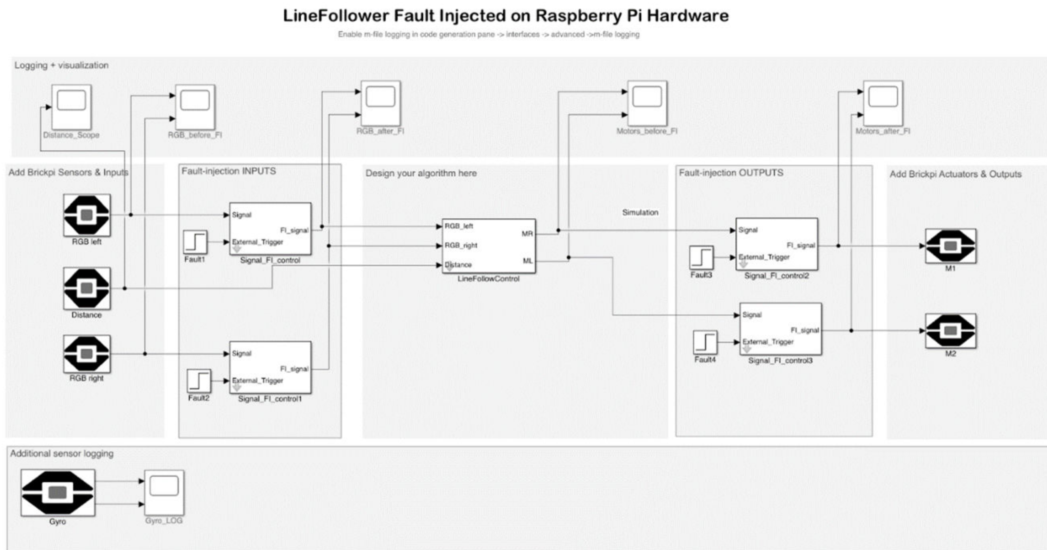

Figure 2: The control model of the robot system

The framework targets an automated build server. We are currently evaluating the appropriate automation servers for which options are Azure® DevOps, CircleCI, Travis CI, or Jenkins with GitHub repo.

Different from related work, the proposed framework is feasible considering its roadmap to automate mutation testing on Simulink software. It also employs an open-source Dextool integrated in continuous integration (CI) and continuous delivery (CD) build pipelines.

4. Pilot Study: Lego Robot System

Robotic systems are a common part of the manufacturing industry today. They automate the applications to reduce costs and time for labor related processes [20]. MATLAB/Simulink supports robot systems such as LEGO MINDSTORMS EV3 containing ARM9-based processors. It also enables engineers to program and run-on LEGO MINDSTORMS EV3 robots [21].

The pilot study called the Lego robot system uses two colour sensors to follow a black line and 1 gyro sensor to stop its motion, when placing a dark surface in front of the gyro sensor, the robot stops. The line is created by placing white tiles with a black line on the floor. Its controller is implemented in MATLAB/Simulink. The controller is run on a RaspberryPI CPU which connects to the actuators and sensors using a BrickPI interface.

Figure 2 shows the control model of the robot, there are different sub models for the BrickPI sensors, actuators, fault injection inputs and fault injection outputs. There are several Simulink scopes that allow one to observe the progression of the different signals generated during the operation of the robot. The Fault injection blocks allow one to set several properties of the fault injection while the block labelled “Design your algorithm here” allows one to set the properties

of the Proportional, Integral, Derivative (PID) control of the robot.

Since the proposed framework applied on C++ code of the Simulink models, we automatically generate C++ code of the control model of the robot system using Simulink Coder package. Then, we employ the framework using Dextool for automatic mutant generation in the second step. The already written unit tests in the first step will be executed on the generated mutants. Dextool provides us desired metrics as being an output of the mutation testing procedure. Thanks to the Simulink Test environment, all these steps run-on an automation framework.

5. Discussion

This section presents a discussion on possible threats to the validity of the current work, provides open research challenges on mutation testing of Simulink models and its software artifacts, and explains further studies within the scope.

5.1. Threats to the Validity

The following subsections discuss threats to internal and external validity. Also, it provides possible mitigation solutions.

5.1.1. Internal Validity

To cope with complexity caused by redundant mutants, Simulink model is a good candidate to apply mutation testing. Because Simulink code is lower level than the model level and contains more structural information. However, the advances of mutation testing provide us with necessary resources to tackle complexity issues of mutation testing, such as redundant mutants. Mutation testing is currently state-of-the-practice rather than state-of-the-art [22, 1].

5.1.2. External Validity

The current work addresses unit level faults rather than model level or integration faults. However, we plan to integrate our framework on model level to address higher level faults. Moreover, there is high correlation between model blocks, their integration, and their corresponding codes. Within the scope of further experimentation, we plan to investigate their correlation to address higher level faults by applying lower-level mutation testing.

5.2. Open Research Challenges and Further Works

There are various challenges and opportunities while applying mutation testing on Simulink models are its corresponding C++ programs. They can be separated based on the artifacts applied on.

On C++ programs, the mutation operators are mostly general-purpose and covering conventional usage of the programming languages. The Simulink models utilized for embedded systems are more specific and demanding definite mutation operators. For example, C++ on Simulink is not case sensitive, while general purpose C++ is case sensitive. Readers may refer [23] for more details between C++ and MATLAB. Therefore, the mutation operators addressing

case sensitivity become irrelevant and cause redundant mutants. Fault models for systems modelled by Simulink are also specific, and thus corresponding tests become more explicit than general-purpose C++ programs. To this end, the definition of explicit mutation operators and preparation of tests addressing specific fault models become more challenging and provide us many opportunities.

On Simulink models, there is not any consensus which mutation operators and how mutation testing will be applied based on the provided related works. Except Phil et al. proposed an open-source toolset called SIMULTATE [17], there are also missing comprehensive works on automatization of the mutation testing. Already proposed works attempt to cover entire blocks on Simulink models by using generic mutation operators. However, tests aim to cover each specified block. Therefore, the definition of mutation operators for different types of model blocks needs to be more specific. Based on these challenges for mutation testing on Simulink models, there are many research and development opportunities.

Since we address only automatization of mutation testing on Simulink code integrated in state-of-the-art continuous integration (CI) and continuous delivery (CD) build pipelines, we keep the above-mentioned challenges for further studies. The utilized open-source Dextool provides us with a selection of desired mutation operators to experiment their effectiveness on the case study.

6. Conclusion

In this paper we argued the need for model-based mutation testing, in the context of MATLAB/Simulink. Based on an overview of related work, we conclude that such a tool is feasible, however that there is not yet a consensus on an effective suite of mutation operators. No study has been proposed on the application of mutation testing to Simulink software artifacts. This work aims to fill this gap. We therefore propose a proof-of-concept tool (currently in the design phase) that would integrate Dextool, MATLAB/Simulink and a continuous integration build server. We intend to validate the tool on a line follower robot, implemented in Lego Mindstorms controlled by a MATLAB/Simulink model. Moreover, we discuss open research challenges and further studies including possible threats to the validity of the current work.

Acknowledgments

The authors express their gratitude to anonymous reviewer for their valuable comments and suggestions on earlier versions of the current paper. This work is supported by the Flanders Innovation and Entrepreneurship under grant number HBC.2021.0010 entitled EFFECTS.

References

[1] A. Parsai, S. Demeyer, Comparing mutation coverage against branch coverage in an industrial setting, International Journal on Software Tools for Technology Transfer 22 (2020) 365–388.

[2] R. G. Hamlet, Testing programs with the aid of a compiler, IEEE transactions on software engineering (1977) 279–290.
[3] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (1978) 34–41.
[4] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE transactions on software engineering 37 (2010) 649–678.
[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in Computers, volume 112, Elsevier, 2019, pp. 275–378.
[6] R. N. Charette, This car runs on code, IEEE spectrum 46 (2009) 3.
[7] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, H. Brandl, Momut:: Uml model-based mutation testing for uml, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–8.
[8] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, B. V. Schmidt, Model-based mutation testing of an industrial measurement device, in: International Conference on Tests and Proofs, Springer, 2014, pp. 1–19.
[9] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, W. E. Wong, Model-based mutation testing—approach and case studies, Science of Computer Programming 120 (2016) 25–48.
[10] O. Kilincceker, A. Silistre, F. Belli, M. Challenger, Model-based ideal testing of gui programs–approach and case studies, IEEE Access 9 (2021) 68966–68984.
[11] O. Kilincceker, E. Turk, F. Belli, M. Challenger, Model-based ideal testing of hardware description language (hdl) programs, Software and Systems Modeling (2021) 1–32.
[12] Y. Zhan, J. A. Clark, Search-based mutation testing for simulink models, in: Proceedings of the 7th annual conference on Genetic and evolutionary computation, 2005, pp. 1061–1068.
[13] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, G. Weissenbacher, Mutation-based test case generation for simulink models, in: International Symposium on Formal Methods for Components and Objects, Springer, 2009, pp. 208–227.
[14] N. T. Binh, et al., Mutation operators for simulink models, in: 2012 Fourth International Conference on Knowledge and Systems Engineering, IEEE, 2012, pp. 54–59.
[15] L. T. M. Hanh, N. T. Binh, K. T. Tung, A novel test data generation approach based upon mutation testing by using artificial immune system for simulink models, in: Knowledge and Systems Engineering, Springer, 2015, pp. 169–181.
[16] M. Stephan, M. H. Alalfi, J. R. Cordy, Towards a taxonomy for simulink model mutations, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2014, pp. 206–215.
[17] I. Pill, I. Rubil, F. Wotawa, M. Nica, Simultate: A toolset for fault injection and mutation testing of simulink models, in: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2016, pp. 168–173.
[18] A. Parsai, S. Demeyer, S. De Busser, C++ 11/14 mutation operators based on common fault patterns, in: IFIP International Conference on Testing Software and Systems, Springer, 2018, pp. 102–118.
[19] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, J. J. Domínguez-Jiménez, Assessment of class mutation operators for c++ with the mucpp mutation system, Information and Software Technology 81 (2017) 169–184.

[20] www.robots.com, Three types of robotic systems, 2021. URL: https://www.robots.com/articles/three-types-of-robotic-systems.
[21] E. Schoofs, J. Kisaakye, B. Karaduman, M. Challenger, Software agent-based multi-robot development: A case study, in: 2021 10th Mediterranean Conference on Embedded Computing (MECO), IEEE, 2021, pp. 1–8.
[22] H. Coles, T. Laurent, C. Henard, M. Papadakis, A. Ventresque, Pit: a practical mutation testing tool for java, in: Proceedings of the 25th international symposium on software testing and analysis, 2016, pp. 449–452.
[23] Mathworks, Comparison of matlab and other oo languages, 2021. URL: https://nl.mathworks.com/help/matlab/matlab_oop/matlab-vs-other-oo-languages.html.