MUT4SLX : fast mutant generation for Simulink

MUT4SLX: Fast Mutant Generation for Simulink

Halil Ibrahim Ceylan*, Onur Kilincceker†, Mutlu Beyazıt†, Serge Demeyer†
*Universiteit Antwerpen †Universiteit Antwerpen and Flanders Make

***Abstract*—Several experience reports illustrate that mutation testing is capable of supporting a "shift-left" testing strategy for software systems coded in textual programming languages like C++. For graphical modelling languages like Simulink, such experience reports are missing, primarily because of a lack of adequate tool support. In this paper, we present a proof-of-concept (named MUT4SLX) for automatic mutant generation and test execution of Simulink models. MUT4SLX features 15 mutation operators which are modelled after realistic faults (mined from an industrial bug database) and are fast to inject (because we only replace parameter values within blocks). An experimental evaluation on a sample project (a Helicopter Control System) demonstrates that MUT4SLX is capable of injecting 70 mutants in less than a second, resulting in a total analysis time of 8.14 hours.**

***Index Terms*—software testing; mutation testing; mutation analysis; Simulink models; cyber-physical systems**

I. Introduction

"Shift-left" is a commonly adopted paradigm in the automated software testing community, emphasising that tests should be executed against the system under test as early as possible [1]. In this paradigm, test suites are expected to be strong, catching defects before they are deployed into production. To measure the strength of a test suite, code coverage (i.e. statement, branch, ...) is commonly used, although it is known to be a poor indicator of the actual strength of a test suite. In the academic literature *mutation testing* is acknowledged as the state-of-the-art technique to assess the fault detection capacity of a test suite [2]. The technique injects artificial faults in the system under test (based on a list of *mutation operators*), subsequently executing the test suite to see whether it is strong enough to catch the injected fault [3]. Several experience reports illustrate that mutation testing is capable of supporting a "shift-left" testing strategy for software systems coded in textual programming languages like C++ [4], [5]

Today, MathWorks Simulink ® is the go-to platform for model-based development of cyber-physical systems [6]. Simulink models the system under test at another level of abstraction using blocks that transform input signals into output signals, as such modelling the behaviour of the cyber-physical system under implementation. Engineers simulate the behaviour in the Simulink environment manually verifying whether the final output signal matches the expected outcome. Once the model satisfies the requirements, code is generated to be deployed on a real-time embedded platform.

Today, Simulink is equipped with two different test execution environments – Signal Builder and Simulink Test. In both these environments, facilities are available to assert whether the output signal is the same as the expected baseline signal, as such imitating the fully automated xUnit behaviour of passing (green) or failing (red) tests. This permits to adopt a *"model-based shift left"* testing strategy for engineers that wish to do so [7]. Tools that measure the coverage of a given test suite are provided as well, counting how many of the blocks and signals have been exercised by the test suite. However, just like code coverage, block coverage is a poor indicator of the strength of a test suite. Mutation testing for Simulink models would offer a better way to measure the fault detection capacity of a test suite.

Consequently, we set out to explore mutation testing in the context of Simulink. We had access to a quality assurance team within a company that embraced modelling as the central activity in their development process. The company combines model driven development with a continuous integration build pipeline, using an extensive suite of regression tests executed via Signal Builder with output assertion blocks. As part of a "model based shift-left" strategy the company asked us to provide tool support for monitoring the strength of their regression test suite. If possible this support should come in the form of a fully automated tool to be integrated into the continuous integration build pipeline.

Based on a literature survey, we identified two proof-of-concept mutation testing tools: SIMULTATE [8] and FIM [6]. These tools defined a comprehensive set of mutation operators based on the grammar of Simulink and demonstrated that one can alter a given model via the API of the Simulink environment. Nevertheless, test execution was lacking in both tools and was a necessary prerequisite for our project. Therefore we created MUT4SLX (Mutation Testing For Simulink) a proof-of-concept tool which builds upon the experience from previous research.

MUT4SLX is published under an open-source licence[1] and provides the following features.

- 15 mutation operators which are modelled after realistic faults mined from an industrial bug database.
- Fast mutant generation by manipulating parameters of Simulink blocks rather than replacing the entire block.
- Automatic test execution via Signal Builder and Simulink Test.
- Extensive test report with pie charts illustrating the killed and survived mutants.

The remainder of this paper is organised as follows. Section II gives an overview of the state of the art, in particular two earlier proof-of-concepts which inspired our work (SIMULTATE and FIM). Section III describes the design of MUT4SLX, which is validated in Section IV among other comparing against FIM. We summarize our conclusions in Section V.

[1]The tool is available at: https://github.com/haliliceylan/MUT4SLX/

II. Related Work

Today, there is little research on mutation testing for Simulink models as witnessed by a recent systematic literature survey conducted by Papadakis et al [2]. This can be expected because Simulink is very specific to particular application domains, such as automotive and avionics. However, those areas require a high degree of reliability and trust and their development is based on stringent safety standards. So one may expect that mutation testing will eventually be adopted in this context as well. Based on a literature survey, we identified three relevant papers, all of which on technology readiness levels 1 (technology concept formulated) or 2 (experimental proof of concept).

The first report we found is a paper by Hanh and Bihn which specified six categories of mutation operators varying from type to expression operators [9]. These mutation operators are defined based on the grammar of Simulink (mutating signals and blocks) and is such are quite comprehensive. No analysis is made based on a fault model of frequently occurring faults in realistic applications. The paper reports on applying these models to a quadratic model and reports mutant execution results.

A second report concerns the SIMULTATE tool which was introduced to automate mutant *generation* for Simulink models [8]. The tool employs a Python API for injecting the faults into model blocks by means of an interactive user interface. For the mutant generation SIMULTATE injects all mutants in a single pass; but each mutant should then be enabled or disabled individually. Compared to the earlier work of Hanh and Bihn, SIMULTATE supports mutating signals only. There is no automated test execution; engineers are expected to manually verify whether the mutated output signal is different from the original one.

FIM is the third proof-of-concept tool we have identified [6]. FIM inherits mutation operators of SIMULATE tool and supports ten operators for signals and five operators for blocks. The tool mainly focuses on safety analysis via extensive fault injection capabilities. The tool provides single and multi-mode options for generating a single model with multiple faults or multiple models with a single fault. It uses the MATLAB command line interface to trigger the tool and a custom library from which block can be changed easily. As such, the mutant generation is faster than SIMULTATE as it does not trigger the MATLAB user interface. FIM is evaluated with an Aircraft Elevator Control System (AECS) from the avionics-aerospace domain. The paper only reports the generation of mutants, not their execution.

The state-of the art has specified a comprehensive suite of mutation operators based on the Simulink grammar, mutating signals and blocks. These mutation operators are not yet refined based on a fault model of frequently occurring faults in realistic applications. Proof-of-concept tools adopted different techniques to generate mutations, either via the MATLAB API (accessible via Python) or via the command line.
Automated test execution is lacking in all tools, mainly because the diversity in test execution engines for Simulink.

III. MUT4SLX Tool Architecture

The design and implementation MUT4SLX is the result of a joint research project with an industrial partner. The industrial partner is a medium-sized enterprise which is experienced in model-based design and development. The company was in search of an effective approach for model-based shift left testing on cyber-physical systems. Through this collaboration, we were able to gain access to their expertise in Simulink development and learn about the real-world challenges they face in model-based testing. This allowed us to design and develop MUT4SLX with a better understanding of the needs and requirements of practitioners in the field. The collaboration also provided us with access to a large and complex Simulink model, which we used to test and validate the performance and effectiveness of the tool. Furthermore, working with an industrial partner made it possible to obtain feedback and suggestions for improvement from industrial experts who use model-based testing tools on a daily basis. This helped us to identify and address potential issues and limitations of the tool, as well as to add new features and functionalities that are most relevant and useful to their practice. Overall, the collaboration with the company was essential to the successful implementation of MUT4SLX.

Most importantly, we defined the mutation operators in close collaboration with our industrial partner. We first formulated mutation operators associated with the bugs in their database. Later, we also added other mutation operators which were deemed relevant to the ones we formulated by the industrial experts. To prioritize these operators, we considered the frequency of the occurrences of the bugs associated with them. We then conducted several manual trials to arrive at a set of mutation operators (see Table I) that were suitable for our needs. A detailed description of the semantics for the mutation operators can be found on the tool web-site [https://github.com/haliliceylan/MUT4SLX/].

Name	Description
ROR	Relational Operator Replacement
LOR	Logical Operator Replacement
ASR	Arithmetic Sign Replacement
MMR	Min-Max Replacement
ICR	If Condition Replacement
TOR	Trigonometric Operator Replacement
MOR	Math Operator Replacement
PMR	Product Multiplication Replacement
POR	Product Operator Replacement
FIR	For Index Replacement
FLR	For Limit Replacement
UDO	Unit Delay Operation
STR	Switch Threshold Replacement
SCR	Switch Criteria Replacement
CR	Constant Replacement

Note that MUT4SLX was designed to easily provide additional mutation operators; the open source license is an extra incentive for users who wish to provide extensions.

TABLE I
Mutation Operators included in MUT4SLX

Figure 1 depicts the architecture of MUT4SLX, which comprises two main components: mutant generation and mutant execution. The following subsections elaborate more on these main components.

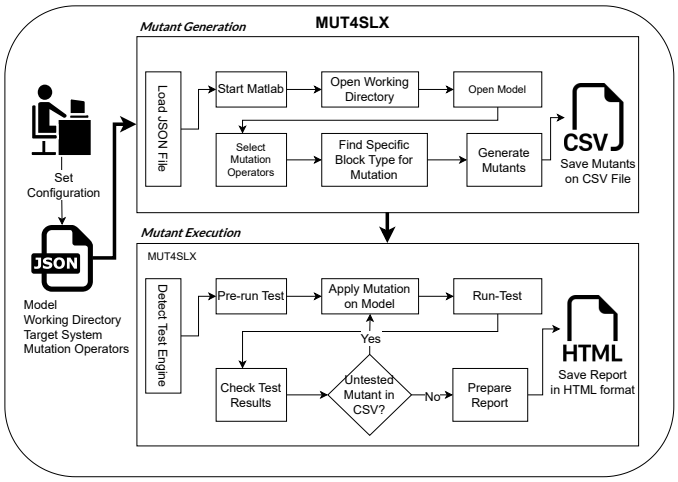

Fig. 1. Overview of MUT4SLX

A. Mutant Generation

The mutant generation component is responsible for reading relevant parameters from the configuration and loading the model, as well as identifying the possible mutants. Users can use the CSV output generated by this component to preview the number of mutants, ignore certain mutants, or prioritize specific mutants before mutant execution takes place. Additionally, users can observe how newly added mutation operators create mutations in specific blocks. The configuration file can also be modified to target only specific subsystems, and multiple Simulink models containing multiple models can be mutated. Since only possible mutations are listed without altering the blocks during mutant generation, the component can operate efficiently without reloading the model repeatedly or waiting for any write operations.

B. Mutant Execution

The mutant execution component executes the given test cases on each of the mutants to count the number of killed and surviving mutants. First, it loads the model into memory by opening the associated file. Subsequently, it attempts to detect the test system used by the model. Currently, two test engines are supported: Signal Builder and Simulink Test.

After detecting the test engine, a pre-run of the tests is performed on the original model to ensure that all tests pass. Then, for each row in the CSV file created in the previous step, the corresponding mutant is injected and the test suite is executed. In this process, the original model remains loaded into memory and the relevant block and parameter are mutated. Only one mutation is applied at a time, and only one parameter is changed in each block. After running the test on the generated mutant, the relevant parameter in the relevant block is reverted to its original state; that is, the original model is recovered. We chose this approach because modifying a block's parameter is faster than replacing the whole block, and reverting back the mutation to recover the original model is faster then reopening the original model. Thus, mutants can be generated and tested very quickly without waiting for the model to close and reload into memory for each mutation. When all the mutants in the CSV file are exhausted, a report is generated and written into an HTML file. The report (see Fig. 2) contains different visualizations (pie charts etc) to reflect the collected data and the results.

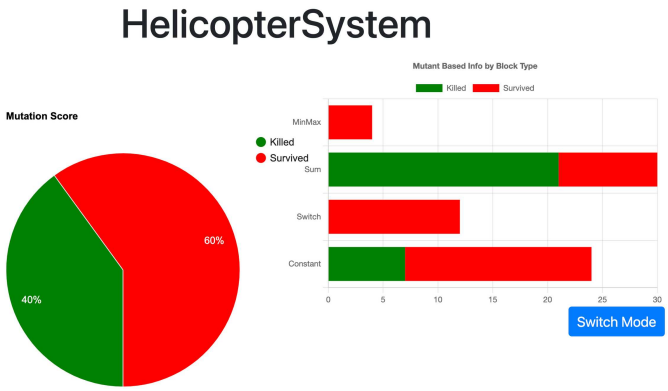

Fig. 2. A Screenshot of Test Report of MUT4SLX

IV. Evaluation

We evaluate MUT4SLX tool by comparing our results with FIM on three simulink models available on the internet, where only one contains an automated test suite. Table II provides descriptive statistics for each model.

System	Number of Subsystem	Total Block Count
Helicopter	11	301
Aircraft Elevator	37	825
Automatic Transmission	17	65

TABLE II
DESCRIPTIVE STATISTICS FOR THE PROJECTS UNDER INVESTIGATION (LAST ACCESSED ON 19.05.2023)

Helicopter Control System. This is a demonstration control system for applying model-based design to specific certifications using the MathWorks toolbox. It provides a workflow that is in line with the guidelines set out in ARP4754A, DO-178C, and DO-331 certifications [10]. Its purpose is to demonstrate how these guidelines can be implemented in a practical context to ensure safety and compliance in aviation software development. It is a large system consisting of 11 different subsystems, each of which is used multiple times within the overall structure, and a total of 301 Simulink blocks. The model contains 6 tests to verify different requirements using the Simulink Test framework. These tests are divided into three sections, focusing on testing the three axes of flight for the helicopter system: yaw, pitch, and roll. The maximum execution time for all 6 tests on the system is 6.5 minutes. During these tests, simulations are performed for each of the three axes. For more details on Helicopter Control System, we refer the interested reader to [10].

Aircraft Elevator Control System. This is one of the projects used in the validation of FIM tool [11]. An aircraft elevator serves as a key flight control surface responsible for maneuvering the aircraft along the lateral axis. This model does not contain any tests.

Automatic Transmission Controller System is a system that electronically controls the shifting of gears in an automatic

Tool	Model	Number of Mutants	Killed Mutants	Mutation Score	Mutant Generation Time	Mutant Execution Time
MUT4SLX	Helicopter	70	28	40%	$<$ 1 sec	8.14 hours
MUT4SLX	Aircraft	205	-	-	12.2 secs	-
MUT4SLX	Autotransmission	17	-	-	4.8 secs	-
FIM	Helicopter	-	-	-	-	-
FIM	Aircraft	41	-	-	74.3 secs	-
FIM	Autotransmission	13	-	-	20.5 secs	-

TABLE III
COMPARISON RESULTS FOR MUT4SLX AND FIM

transmission, optimizing performance without requiring manual input from the driver. This model is also used in [11] and does not contain any tests.

We use the Helicopter model to assess the full functionality of our tool. The Aircraft and Autotransmission models are used to compare our tool with FIM tool. Results are given in Table III. Note that the Aircraft and Autotransmission models have no test cases and FIM has no test execution capability; therefore, there is no data related to mutant execution for them. Also, there is no data on Helicopter using FIM tool because we were not able to run the tool on Helicopter model.

The validation starts with mutant generation to discover the mutants that can be automatically generated from the original model. As shown in Table III, MUT4SLX generates 70 mutants in less than a second.

After mutant generation, the tool triggers the mutant execution. For each mutant, the actual mutant is generated and tests are executed on the mutant until one fails or all passes. After test execution, the tool generates a test report in HTML format. Our observations show that mutant execution takes about 8 hours in total. Interested readers may refer to our tool repository for the test report of the Helicopter model.

Our findings suggest that MUT4SLX successfully identifies mutants very fast for the mutation operators presented in Section 2. However, mutant execution takes considerably longer due to the nature of Matlab simulations. As future work, we plan to reduce mutant execution time by running the tests on multiple mutants in parallel within our tool.

To compare our tool with FIM [6], we tried both tools on the Aircraft and AutoTransmission models. The results of the comparison are presented in Table III.

FIM generates 41 and 13 mutants for the Aircraft and Autotransmission models respectively, whereas MUT4SLX generates 205 mutants for Aircraft and 17 mutants for Autotransmission due to support for increased number of mutation operators. When the block count increases, the performance advantage of MUT4SLX for mutant generation becomes more apparent as presented in Table III such as the Aircraft model. Besides increasing the number of mutants, MUT4SLX is also faster than FIM for mutant generation. For example, MUT4SLX generates 205 mutants in 12.2 seconds, and FIM generates 41 mutants in 74.3 seconds. The main reason for the performance difference is that MUT4SLX modifies the parameters of Simulink blocks whereas FIM uses a custom library to replace them with faulty blocks.

V. CONCLUSION

In this paper, we present MUT4SLX, a proof-of-concept mutation testing tool for Simulink models. MUT4SLX automates mutant generation and mutant execution, and generates detailed test reports in HTML format. As such the tool can be incorporated in a continuous integration build pipeline.

MUT4SLX is validated on three Simulink models available on the internet. The most comprehensive evaluation on a model for a Helicopter Control System shows that MUT4SLX is capable of injecting 70 mutants in less than a second, resulting in a total analysis time of 8.14 hours.

VI. ACKNOWLEDGMENTS

This work is supported by (a) the Flanders Innovation & Entrepreneurship (VLAIO) under grant number HBC.2021.0010 entitled "EFFECTS"; (b) the Research Foundation Flanders (FWO) under grant number S000323N entitled "Basecamp Zero"; (c) the ITEA4 Project "SmartDelta"; (d) Flanders Make, the strategic research centre for the manufacturing industry.

REFERENCES

[1] L. Smith, "Shift-left testing," *Dr. Dobb's Journal*, vol. 26, no. 9, pp. 56–ff, 2001.

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: https://doi.org/10.1109/C-M.1978.218136

[4] R. Baker and I. Habli, "An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, Jun. 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.56

[5] S. Vercammen, M. Borg, and S. Demeyer, "Validation of mutation testing in the safety critical industry through a pilot study," in *Workshop Proceedings ICST 2023 (IEEE International Conference on Software Testing, Verification and Validation)*, 2023.

[6] E. Bartocci, L. Mariani, D. Ničković, and D. Yadav, "Fim: Fault injection and mutation for simulink," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1716–1720. [Online]. Available: https://doi.org/10.1145/3540250.3558932

[7] D. Firesmith, "Four types of shift left testing," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Mar 2015, accessed: 2023-May-9. [Online]. Available: http://insights.sei.cmu.edu/blog/four-types-of-shift-left-testing/

[8] I. Pill, I. Rubil, F. Wotawa, and 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)*. Chicago, IL, USA: IEEE, 2016, pp. 168–173.

[9] L. T. M. Hanh and N. T. Binh, "Mutation operators for simulink models," in *2012 Fourth International Conference on Knowledge and Systems Engineering*. Danang, Vietnam: IEEE, 2012, pp. 54–59.

[10] B. Potter, "Helicopter case study for do-178 using mathworks tools," 18.05.2020, accessed on 26.04.2023. [Online]. Available: https://nl.mathworks.com/matlabcentral/fileexchange/56056-do178_case_study

[11] D. Yadav, "Fimtool source," 18.05.2022, accessed on 26.04.2023. [Online]. Available: https://gitlab.com/DrishtiYadav/fimtool