MASS: A tool for Mutation Analysis for Space CPS

Oscar Cornejo
SnT Centre, University of Luxembourg
Luxembourg, Luxembourg
oscar.cornejo@uni.lu

Fabrizio Pastore
SnT Centre, University of Luxembourg
Luxembourg, Luxembourg
fabrizio.pastore@uni.lu

Lionel Briand
SnT Centre, University of Luxembourg
School of EECS, University of Ottawa
Ottawa, Canada
lionel.briand@uni.lu

ABSTRACT

We present *MASS*, a mutation analysis tool for embedded software in cyber-physical systems (CPS). We target space CPS (e.g., satellites) and other CPS with similar characteristics (e.g., UAV).

Mutation analysis measures the quality of test suites in terms of the percentage of artificial faults detected. There are many mutation analysis tools available but they are inapplicable to CPS because of scalability and accuracy challenges.

To overcome such limitations, *MASS* implements a set of optimization techniques that enable the applicability of mutation analysis and address scalability and accuracy in the CPS context. *MASS* has been successfully evaluated on a large study involving embedded software systems provided by industry partners; the study includes an on-board software system managing a microsatellite currently on-orbit, a set of libraries used in deployed cubesats, and a mathematical library provided by the European Space Agency. A demo video of *MASS* is available at https://www.youtube.com/watch?v=gC1x9cU0-tU.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Mutation analysis, CPS, European Space Agency

1 INTRODUCTION

Software has an important role in modern cyber-physical systems (CPS) and in space systems in particular. Indeed, software components are used, for example, to control the system, encapsulate the data, and manage the communication with other systems; similar features are also implemented in other critical CPS such as automotive, avionics, and industry 4.0 (e.g., robots).

The embedded software running on space CPS (hereafter, space software) and similar CPS has to meet strict quality constraints imposed by regulatory agencies (e.g., the European Space Agency - ESA [14]). Software validation and verification (V&V) activities largely rely on test suites, which are usually derived manually from requirements. Unfortunately, the manual definition of test cases may lead to incomplete test suites; similarly, the independent V&V procedures mandated by standards (e.g., ESA regulates Independent Software V&V — ISVV [12, 13]), which are manually performed, provide limited guarantees about the quality of CPS software systems. *Automated means to assess the quality of test suites are therefore necessary to ensure CPS quality* and motivated the project that led to the development of *MASS* [2].

Mutation analysis is an effective way to automatically assess the quality of a test suite; it consists of measuring the proportion of artificially injected faults detected by a test suite [10]. Despite its potential, mutation analysis is not widely adopted in industry because of its limited scalability and doubts about the pertinence of the mutation score as adequacy criterion [25]. For example, space software is typically large and accompanied by test suites that take a long time to execute, which leads to a large number of mutants that may require months to be tested if scalable solutions are not in place. The literature about mutation analysis has proposed a number of optimizations to overcome the problems presented above. On one hand, scalability problems can be addressed by sampling the mutants [19, 33], or by prioritizing and selecting the test cases to be executed for each mutant [34]. On the other hand, equivalent and redundant mutants can be identified by means of trivial compiler optimisations [23], or by comparing the code coverage of the original program against its mutants [20, 28–30]. Nevertheless, none of these techniques and tools have been assessed in industrial contexts and, furthermore, there are no studies about the integration of such optimizations and their combined benefits.

In this paper, we introduce *MASS* (Mutation Analysis for Space Software), a tool for the assessment of test suites based on mutation analysis. *MASS* integrates a pipeline of solutions that make mutation analysis feasible with large software systems. The three main features of MASS are (1) the automated identification of equivalent mutants using an ensemble of compiler optimization options, (2) the computation of the mutation score based on mutant sampling with a fixed size confidence interval approach (FSCI), (3) the automated identification of likely equivalent mutants based on code coverage. Furthermore, *MASS* provides information useful to produce a verification report for ISVV activities; it includes the sets of live mutants and killed mutants (i.e., mutants that are discovered by the test suite), the statement coverage of the test suites under analysis, and the mutation score (i.e., the percentage of mutants discovered by the test suite).

We empirically evaluated the scalability and accuracy of *MASS* with case study subjects provided by our industry partners, which are ESA, GomSpace Luxembourg (GSL), a manufacturer and supplier of nanosatellites [18], and LuxSpace (LXS), a developer of infrastructure products (e.g., microsatellites) and solutions for space [24]. Not only *MASS* enabled the identification of shortcomings affecting the test suites of these software systems but, furthermore, we demonstrated that mutation analysis was indeed feasible in realistic industrial contexts. In short, mutation analysis with *MASS* can be completed in few days, even for large systems, which enables its adoption in ISVV contexts.

*Paper submitted for peer review to ICSE 2022 - The 44th International Conference on Software Engineering. Do not distribute.

The paper proceeds as follows. Section 2 presents related work. Section 3 describes our mutation analysis pipeline. Section 4 provides details about the *MASS* architecture and availability. Section 5 summarizes our empirical results. Section 6 concludes the paper.

2 RELATED WORK

Mutation analysis is a topic that has been extensively discussed over the years in the literature [26]. The mutation testing tool repository refers to 87 mutation analysis tools [4]; however, only a small portion of them can be applied to space software and related CPS, which are typically implemented with Ada, C, and C++ [1, 5, 9, 11, 21, 22, 27, 32]. Furthermore, only three of these tools are still under active maintenance [1, 5, 11]. Finally some of these tools (i.e., Mull [11], Dextool [1], Accmut [32], Mart [5]) require the software under test (SUT) to be compiled as LLVM bitcode, which prevents their applicability to a wide range of CPS software because (a) CPS software often relies on compiler optimizations not supported by the LLVM infrastructure and (b) there is no guarantee that the software artifacts compiled with LLVM are equivalent to those compiled with the original compiler (e.g., LLVM is not qualified by ESA/ECSS for category A software [15]). Also, some of these tools apply mutations dynamically, which is infeasible for CPS software that runs on dedicated simulators.

The few tools that do not rely on LLVM and are thus widely applicable to CPS software (i.e., Milu [22] and SRCIRor [21]) either require the generation of preprocessed source code [22], which leads to a large number of compilation problems with large software systems, or implement a limited set of mutation operators and do not detect equivalent and redundant mutants based on compiler optimization techniques [21].

Based on the above, we conclude that there is a lack of tools applicable to large software systems for CPS. To overcome the limitations above, *MASS* mutates the source code and relies on the original compiler infrastructure. Also, it relies on compiler optimizations for detecting equivalent and redundant mutants. Finally, it is the first tool to make mutation analysis scalable thanks to the integration of both mutants sampling and test cases selection and prioritization.

3 MASS METHODOLOGY

MASS is the tool supporting our methodology for the mutation analysis of embedded software within CPS [8]. *MASS* performs mutation analysis in eight steps.

In *Step 1*, *MASS* collects the SUT code coverage. Code coverage enables some optimizations such as not mutating statements that are not covered by the test suite, and executing only the test cases that cover a mutated statement.

In *Step 2*, *MASS* generates mutants by relying on an extended sufficient set of operators, which consists of ABS, AOR, ICR, LCR, ROR, SDL, UOI, AOD, LOD, ROD, BOD, SOD, and LVR [8].

In *Step 3*, *MASS* compiles the mutants in an iterative way to leverage the incremental compilation implemented by build systems. It compiles every mutant within the same source folder structure; for each mutant, it replaces the corresponding original source file with the mutated one and builds the software. The original file is restored after each compilation. This enables the reuse of compiled objects thus saving a considerable amount of time.

In *Step 4*, *MASS* removes equivalent and redundant mutants from the set of generated mutants by relying on compiler optimizations (i.e., O0, O1, O2, O3, O4, Ofast, Os for the GCC compiler [3]). For every optimisation level, *MASS* re-compiles every mutant and stores the SHA-512 hash of the generated executable. Equivalent and redundant mutants are identified by comparing SHA-512 hashes, which is more efficient than comparing the compiled executables.

To further address scalability issues, in *Step 5*, *MASS* samples mutants from the set of compiled, nonequivalent, and nonredundant mutants. *MASS* supports *proportional uniform sampling* [33], *proportional method-based sampling* [33], *uniform fixed-size sampling* [19], and *FSCI-based sampling*. *Proportional uniform sampling* randomly samples a user-specified percentage of mutants, *proportional method-based sampling* randomly samples a user-specified percentage of mutants for each method of the SUT, *uniform fixed-size sampling* randomly samples a user-specified number of mutants across the whole program. *FSCI-based sampling* (hereafter, FSCI) determines the sample size dynamically while exercising mutants, based on a fixed-width sequential confidence interval approach [17]. With FSCI, *MASS* iteratively selects a random mutant and exercises it with the SUT test suite; the process stops when the confidence interval computed with the Clopper-Pearson method [6] is below a user-specified threshold (defaults to 0.10). Since FSCI enables *MASS* to provide statistical guarantees about the accuracy of the estimated mutation score (see Section 5), we therefore recommend its use. *FSCI* is a novel feature of *MASS*.

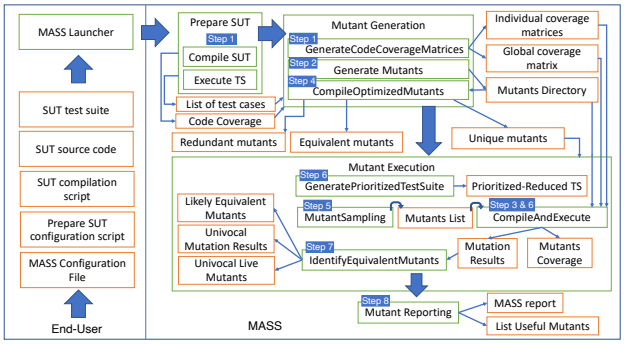
In *Step 6*, *MASS* executes a prioritized and reduced set of test cases for each mutant. First, *MASS* selects only the test cases that cover the mutated statement. Second, *MASS* defines the order of execution of test cases based on the likelihood of killing a mutant. To determine this likelihood, we rely on code coverage to determine how dissimilar two test cases are and compare the number of times each statement has been covered by test cases. To measure the distance between two test cases we use the cosine similarity distance.

In *Step 7*, *MASS* identifies likely equivalent mutants based on code coverage; a mutant is likely equivalent when the cosine similarity distance from the original program is equal to zero; such threshold has been empirically determined [8]. Redundant mutants cannot be identified with this method because it has not been possible to empirically determine a threshold for this purpose [8]; such finding is probably due to test suites being typically unable to distinguish redundant mutants [31].

In *Step 8*, *MASS* estimates the mutation score as the number of killed mutants divided by the number of total mutants, excluding equivalent and redundant mutants. *MASS* also reports other relevant metrics such as statement coverage, the number of executed mutants, and the number of killed and live mutants.

4 TOOLSET ARCHITECTURE

We implemented *MASS* with C, Python, and Bash. Figure 1 shows the architecture of *MASS*; it consists of five components: *Launcher*, *Prepare SUT*, *Mutant Generation*, *Mutant Execution*, and *Mutant*



Legend: Green boxes represent software components. Orange boxes are inputs and outputs. Thick arrows capture control flow. Thin arrows show the data flow.

Figure 1: Architecture of the *MASS* tool.

Reporting. Figure 2 shows the structure of a project analyzed with *MASS*.

The *Launcher* component orchestrates the execution of each step of *MASS*. The inputs to be provided by the end-user are (1) the path to the source code of the SUT, (2) the test suite to evaluate (*SUT Test Suite* in Figure 1), (3) a script with the compilation commands to be used to build the SUT (*SUT compilation script*), (4) a script with the commands required for executing the test suite and collecting code coverage (*Prepare SUT configuration script*), and (5) the *MASS* configuration file, which is used to specify a number of options including the mutants sampling strategy, the execution environment (i.e., single machine or HPC), and the type of test suite prioritization to apply.

The *Prepare SUT* component compiles and executes the SUT test suite to collect code coverage information through *gcov* [16]. For a CPS without a filesystem, we use *GDB* for dumping coverage information at runtime. Then, based on code coverage, the *Prepare SUT* component generates a file that, for every source code statement, reports the test cases that cover the statement; such file is used, later on, to select the test cases to be executed with each mutant.

The *Mutant Generation* component processes the SUT source code and the code coverage files to generate mutants (i.e., it discards mutants for statements that are not covered). Each mutant is univocally identified with a name that captures information about the mutated statement (i.e., applied mutation operator, modified source file, line, and column). The *Mutant Generation* component discards non-compilable mutants and mutants detected as being redundant and equivalent according to the compiler optimization approach [23]. The identifier of the mutants not discarded is reported in the file *unique mutants*. The *Mutant Generation* component also generates, for each mutant, a directory with the mutated source files. To generate mutants, we extended the SRCIRor toolset [21].

The *Mutant Execution* component (1) generates a prioritized and reduced test suite, (2) samples and executes mutants, and (3) identifies likely equivalent mutants based on code coverage.

MASS also supports execution on High-Performance Computing (HPC) infrastructures, which is key for the application of mutation analysis with large projects. End-users can leverage an HPC to parallelize both the execution of mutants and the identification of equivalent and redundant mutants based on compiler optimizations.

MASS_WORKSPACE/

- *mass_conf.sh*: MASS configuration file (i.e., optimizations to be used).
- *sut_compilation_script.sh*: SUT compilation commands.
- *prepare_sut_conf_script.sh*: Test suite execution commands, code coverage collection commands.
- *Launcher.sh*: MASS single launcher, the script executes all the steps of the methodology.
- **MASS_STEPS_LAUNCHERS/**: MASS individual launchers (e.g., GenerateMutants.sh).
- **COVERAGE_FILES/**:
  - contains the coverage files, the coverage matrices, and the list of test cases.
- **SRC_MUTANTS/**:
  - contains all the mutant sources, it contains one dedicated folder for each source file.
- **COMPILED_TCE/**:
  - contains the mutants' hashes, it also reports the list of equivalent and redundant mutants.
- **PRIORITIZED_TS/**:
  - contains one file with the prioritized and reduced test suites, and one file with the prioritized test suite.
- **MUTATION/**:
  - contains the list of tested mutants, the mutation traces, the mutants coverage, and the list of killed, live mutants.
- **DETECTION_EQUIVALENTS/**:
  - contains the list of likely equivalent mutants, and the filtered mutation traces (i.e., without equivalent mutants).
- **RESULTS/**:
  - contains the MASS mutation analysis report, and the list of useful mutants.

Figure 2: Structure of a *MASS* project.

Finally, the *Mutant Reporting* component collects all the results of the mutation analysis process and produces a report file (i.e., MASS *report*) with the following data: mutation score, number of killed and live mutants, sampling strategy, total execution time, code coverage. Furthermore, it generates a file with a subset of the live mutants that should be inspected by engineers to improve the test suite (i.e., to generate test cases that kill them). Our objective is to minimize the number of redundant mutants inspected; indeed, the file includes only live mutants that differ from each other in terms of code coverage. Also, since engineers may only be able to inspect the first items on the list, to minimize the number of equivalent mutants inspected, *MASS* sorts the mutants according to their distance from the original SUT (mutants that largely differ appear first since they are unlikely to be equivalent).

The *MASS* toolset and its specifications are available online [7].

5 EMPIRICAL EVALUATION

We have applied *MASS* to five software artifacts: a mathematical library provided by ESA (*MLFS*), a subset of the control software of *ESAIL* (hereafter, $ESAIL_S$), which is a micro-satellite developed by LXS, the libraries *LIBU*, *LIBN*, and *LIBP*, which are developed by GSL and used in cubesat constellations. *LIBN* is a network protocol library. *LIBP* is a light-weight parameter system. *LIBU* is a utility library providing cross-platform APIs [8].

Details about the different artifacts can be found in Table 1. For $ESAIL_S$, we focused on its system test suite executed in the Software Validation Facility (SVF) (i.e., a simulator for the onboard hardware). The other artifacts are tested with either unit or integration test suites. Our case study subjects thus cover different application scenarios for mutation analysis.

Table 1: Case study subjects.

Subject	LOC	Test suite type	# Test cases	Statements coverage
$ESAIL_S$	2 235	System	384	95.36%
LIBN	9 836	Integration	89	63.10%
LIBP	3 179	Integration	170	77.60%
LIBU	10 576	Unit	201	83.20%
MLFS	5 402	Unit	4042	100.00%

Our experiments have shown that identifying equivalent and redundant mutants by combining all the compiler optimizations provided by the GCC compiler is scalable and effective. Indeed, it enables the detection of the largest number of such mutants and can be executed in few hours even for large SUTs. For example, it took only 2 hours to compile the 5 347 mutants generated for $ESAIL_S$, our largest case study subject.

We demonstrated that FSCI-based sampling is the strategy that selects the smallest number of mutants (between 248 and 366 mutants, for each subject), in addition to providing statistical guarantees on the accuracy of mutation score estimates (i.e., the estimated mutation score differs at most by 5% from the actual one).

We also showed that combining test cases selection and prioritization with FSCI further reduces execution time while still guaranteeing the accurate estimation of the mutation score. For example, for our case study $ESAIL_S$, we reduced mutation analysis time from 11,000 to 1,531 hours. In practice, this makes mutation analysis feasible in seven days with 10 computing nodes. Given the validation procedures for critical CPS are long (e.g., weeks), such execution time is acceptable for both software and ISVV providers. Note that without *MASS* optimizations, mutation analysis would take more than 100 days to complete, even with 100 computing nodes.

Finally, we demonstrated that the strategy adopted by *MASS* to detect nonequivalent mutants based on code coverage leads to extremely accurate results (precision = 81%, recall = 100%). This is important since it increases the chances that the reported live mutants represent actual test suite shortcomings.

6 CONCLUSION

We have presented *MASS*, a tool that makes mutation analysis feasible for space software and, in general, for large embedded software in CPS. Our aim is to support both software developers and regulatory agencies performing independent V&V. The key features of *MASS* include (1) discarding equivalent and redundant mutants through compiler optimizations, (2) generating mutants with a comprehensive set of sufficient mutation operators, (3) accurately sampling mutants with a confidence interval-based approach, (4) reducing the test suite execution time by prioritizing and reducing the number of test cases, and (5) discarding likely equivalent mutants based on code coverage.

We evaluated *MASS* with five representative case study subjects from our industrial partners. Our results show that *MASS* can be effectively applied on large space software; it reduces the processing time of mutation analysis by (1) discarding equivalent and redundant mutants, (2) sampling a subset of the mutants without affecting the accuracy of the estimated mutation score, and (3) prioritizing and reducing test suites. *MASS* is available for download [7].

ACKNOWLEDGMENTS

This work was funded by ESA (ITT-1-9873/FAQAS), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), and NSERC Discovery and Canada Research Chair programs.

REFERENCES

[1] 2021. Dextool. https://github.com/joakim-brannstrom/dextool
[2] 2021. FAQAS project. https://faqas.uni.lu.
[3] 2021. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/
[4] 2021. Mutation Testing Repository. https://mutationtesting.uni.lu/tools.php
[5] Thierry Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: a mutant generation tool for LLVM. In *Proceedings of the 27th ESEC/FSE*.
[6] C. J. Clopper and E. S. Pearson. 1934. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika* (1934).
[7] Oscar Cornejo, Fabrizio Pastore, and Lionel Briand. 2021. MASS toolset. https://github.com/SNTSVV/FAQAS_MASS.
[8] Oscar Cornejo, Fabrizio Pastore, and Lionel Briand. 2021. Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain. *IEEE TSE* (2021).
[9] Márcio Delamaro, José Maldonado, and AP Mathur. 1996. Proteum-a tool for the assessment of test adequacy for c programs user's guide. In *PCS*.
[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* (1978).
[11] Alex Denisov and Stanislav Pankevich. 2018. Mull it over: mutation testing based on LLVM. In *Proceedings of IEEE ICSTW*.
[12] ESA. 2009. ECSS-E-ST-40C - Software general requirements. http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/
[13] ESA. 2017. ECSS-Q-ST-80C Rev.1 - Software product assurance. http://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/
[14] ESA. 2021. European Space Agency. https://www.esa.int
[15] ESA. 2021. RTEMS SMP qualification. https://rtems-qual.io.esa.int/qdp/rtems-smp-status-08042021-final.pdf
[16] FSF Free Software Foundation. 2021. gcov - A Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[17] Jesse Frey. 2010. Fixed-Width Sequential Confidence Intervals for a Proportion. *The American Statistician* (2010).
[18] Gomspace. 2021. Systems for cubesats and nanosatellites. https://gomspace.com/
[19] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *Proceedings of IEEE ISSRE*.
[20] Bernhard JM Grün, David Schuler, and Andreas Zeller. 2009. The impact of equivalent mutants. In *Proceedings of IEEE ICSTW*.
[21] Farah Hariri and August Shi. 2018. SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation. In *ASE*.
[22] Yue Jia and Mark Harman. 2008. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC-PART*.
[23] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE TSE* (2017).
[24] OHB LuxSpace. 2021. The first provider of space systems, applications and services in Luxembourg. https://luxspace.lu/
[25] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *ISSTA*.
[26] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*.
[27] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. 2018. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In *Proceedings of IEEE ICSTW*.
[28] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *Proceedings of the 18th ISSTA*.
[29] David Schuler and Andreas Zeller. 2010. (Un-)covering equivalent mutants. In *Proceedings of IEEE ICST*.
[30] David Schuler and Andreas Zeller. 2013. Covering and uncovering equivalent mutants. *STVR* (2013).
[31] Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2017. A theoretical and empirical study of diversity-aware mutation adequacy criterion. *IEEE TSE* (2017).
[32] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ISSTA*.
[33] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *ASE*.
[34] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of ACM ISSTA*.