MASS: A tool for Mutation Analysis of 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 detected artificial faults. 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

ACM Reference Format:


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 [16]). 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 [14, 15]), 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 [12]. 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 [28]. 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 [21, 36], or by prioritizing and selecting the test cases to be executed for each mutant [37]. On the other hand, equivalent and redundant mutants can be identified by means of trivial compiler optimisations [25], or by comparing the code coverage of the original program against its mutants [22, 31–33]. 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 as adequacy criterion, based on the number of artificially injected faults detected by a test suite, and, furthermore, there are no studies about the integration of such optimizations and their combined benefits.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA  
© 2021 Association for Computing Machinery.  
ACM ISBN 978-1-4503-XXXX-X/18/06. ... $15.00  
https://doi.org/10.1145/1122445.1122456

This paper will appear in the Companion Proceedings of the 44th International Conference on Software Engineering (ICSE2022). IEEE.
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 [19], and LuxSpace (LXS), a developer of infrastructure products (e.g., microsatellites) and solutions for space [27]. 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 a few days, even for large systems, which enables its adoption in ISVV contexts.

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 [29]. The mutation testing tool repository refers to 87 mutation analysis tools [6]; 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, 7, 11, 13, 24, 23, 30, 35]. Furthermore, only three of these tools are still under active maintenance [1, 7, 13]. Finally, some of these tools (i.e., Mull [13], Dextool [1], Acmut [35], Mart [7]) require the software under test (SUT) to be compiled as LLVM bytecode, 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 [17]). 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 [24] and SRIwRor [23]) either require the generation of preprocessed source code [24], 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 [23].

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 [10]. MASS performs mutation analysis in eight steps, which are depicted in Figure 1.

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 [10].

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 [4]). 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 [36], proportional method-based sampling [36], uniform fixed-size sampling [21], and FSCI-based sampling. Proportional uniform sampling randomly samples a user-specified percentage of mutants, proportional method-based sampling randomly samples a user-specified...
To determine this likelihood, we rely on code coverage to deter-
vant metrics such as statement coverage, the number of executed
mutants, and the number of killed and live mutants.

equivalent and redundant mutants. To empirically determine a threshold for
this purpose [10]; such finding
be identified with this method because it has not been possible to
coverage; a mutant is likely equivalent when the cosine similarity
distance. The distance between two test cases we use the cosine similarity
mine how dissimilar two test cases are and compare the number of
execution of test cases based on the likelihood of killing a mutant.

MASS: A tool for Mutation Analysis of Space CPS ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

Figure 2: Architecture of the MASS tool.
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 [18].
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 [8] 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 deter-
mine 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 [10]. Redundant mutants cannot
be identified with this method because it has not been possible to
empirically determine a threshold for this purpose [10]; such finding
is probably due to test suites being typically unable to distinguish
redundant mutants [34].

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 rele-
vant 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. MASS supports
software written in C/C++, built using GCC Make [5] or WAF [26],
and compiled with GCC versions above 4. Furthermore, MASS offers
built-in features to process the SUT test harness Google Test [20].

Figure 2 shows the architecture of MASS; it consists of five com-
ponents: Launcher, Prepare SUT, Mutant Generation, Mutant Execu-
tion, and Mutant Reporting. Figure 3 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 2), (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 in-
cluding 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 [3]. For a
CPS without a filesystem, we use GDB for dumping coverage infor-
mation 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 dis-
cards 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 [25]. The identifier of the mutants not discarded is re-
ported 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 [23].

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.

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 [9].

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, ESAIL5), 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 [10].

Details about the different artifacts can be found in Table 1. For ESAIL5, 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.

Our empirical evaluation concerned (1) the effectiveness of the identification of equivalent and redundant mutants with compiler optimization approaches (MASS Step 4), (2) the accuracy of different mutant sampling techniques (Step 5), (3) the time savings obtained by combining mutants sampling and a reduced and prioritized test suite (Step 6), (4) the accuracy of our approach to detect nonequivalent mutants based on coverage information (Step 7).

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 a few hours, even for large SUTs. The overview of the data collected in our experiments is provided in Table 2. Table 2 shows that it takes approximately two hours to compile the 5,347 mutants generated for ESAIL5, our largest case study subject; also, it takes less than one hour to compile the 28,069 mutants generated for MLFS. The percentage of equivalent and redundant mutants identified by the approach ranges from 23.84% (MLFS) to 38.96% (LIBP).

Table 3 provides information about the number of mutants and the accuracy obtained with the different sampling techniques proposed in the literature (i.e., proportional uniform sampling, and uniform fixed-size sampling) and our approach (i.e., FSCI sampling), across the different subjects. FSCI 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). The sample size obtained with FSCI is much lower than the—worst case—sample size proposed by Gopinath et al. [21], which is 1,000.

Table 4 provides the execution time obtained with the different strategies used in our experiments: (1) testing all the mutants with the original SUT test suite (i.e., traditional mutation analysis), (2) sampling mutants with FSCI and executing them with the original SUT test suite, and (3) sampling mutants with FSCI and executing them with a reduced and prioritized test suite. The data in Table 4 show 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 ESAIL5, we reduced mutation analysis time from 11,000 to 1,865 hours. In practice, this makes mutation analysis feasible in seven days with 10 computing nodes. Given that 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, concerning MASS Step 7, 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)
Table 3: Accuracy with proportional uniform sampling (uniform), uniform fixed-size sampling (fixed) and FSCI sampling.

<table>
<thead>
<tr>
<th>#Mutants</th>
<th>LIBN</th>
<th>LIBP</th>
<th>LIBU</th>
<th>LIBV</th>
<th>MFLP</th>
<th>ESALQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fixed 0.01</td>
<td>fixed 0.01</td>
<td>fixed</td>
<td>fixed</td>
<td>fixed</td>
<td>fixed</td>
</tr>
<tr>
<td>100</td>
<td>1.09</td>
<td>1.09</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>500</td>
<td>2.50</td>
<td>2.50</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>1000</td>
<td>3.50</td>
<td>3.50</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>2000</td>
<td>4.50</td>
<td>4.50</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 4: Execution times (hours) of different strategies.

<table>
<thead>
<tr>
<th>Subject</th>
<th>All mutations</th>
<th>Original test suite</th>
<th>FSTC + Original test suite</th>
<th>FSTC + Test suite reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESALQ</td>
<td>11.00</td>
<td>13.00</td>
<td>14.00</td>
<td>15.00</td>
</tr>
<tr>
<td>LIBN</td>
<td>70.22</td>
<td>70.22</td>
<td>70.22</td>
<td>70.22</td>
</tr>
<tr>
<td>LIBP</td>
<td>13.32</td>
<td>13.32</td>
<td>13.32</td>
<td>13.32</td>
</tr>
<tr>
<td>LIBU</td>
<td>34.54</td>
<td>34.54</td>
<td>34.54</td>
<td>34.54</td>
</tr>
<tr>
<td>MFLP</td>
<td>47.72</td>
<td>47.72</td>
<td>47.72</td>
<td>47.72</td>
</tr>
</tbody>
</table>

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

[8] C. J. Cooper and E. S. Pearson. 1934. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. Bioometrika (1934).
[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.