

# Applicability of Mutation Testing to Flight Software (FAQAS)

## Executive Summary Report (ESR)

F. Pastore, O. Cornejo, E. Viganò

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

ITT-1-9873-ESA-FAQAS-ESR

Issue 1, Rev. 2

November 17, 2021

EUROPEAN SPACE AGENCY. CONTRACT REPORT.

The work described in this report was done under ESA contract.

Responsibility for the contents resides in the author or organisation that prepared it.

The copyright in this document is vested in the University of Luxembourg.

This document may only be reproduced in whole or in part, or stored in a retrieval system, or transmitted in any form, or by any means electronic, mechanical, photocopying or otherwise, either with the prior permission of the University of Luxembourg or in accordance with the terms of ESTEC Contract No. 4000128969/19/NL/AS.

## 1 INTRODUCTION

From spacecrafts to ground stations, software has a prominent role in space systems; for this reason, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Mission failures due to insufficient software sanity checks [14] are unfortunate examples, pointing to the necessity for systematic and predictable quality assurance procedures in space software.

Since one of the primary objectives of software testing is to identify the presence of software faults, an effective way to assess the quality of a test suite consists of artificially injecting faults in the software under test and verifying the extent to which the test suite can detect them. This approach is known as *mutation analysis* [8]. In mutation analysis, faults are automatically injected in the program through automated procedures referred to as mutation operators. Mutation operators enable the generation of faulty software versions that are referred to as *mutants*. Mutation analysis helps evaluate the effectiveness of a test suite, for a specific software system, based on its mutation score, which is the percentage of mutants leading to test failures. Also, mutation analysis enables *mutation testing*, which concerns the automated generation of test cases that discover mutants.

Despite its potential, mutation analysis is not widely adopted by industry. The main reasons include its limited scalability and the pertinence of the mutation score as an adequacy criterion [24]. Indeed, for a large software system, the number of generated mutants might prevent the execution of the test suite against all the mutated versions. Also, the generated mutants might be either semantically equivalent to the original software [21] or redundant with each other [27]. Equivalent and redundant mutants may bias the mutation score as an adequacy criterion. Finally, test generation approaches are preliminary and cannot be applied in industrial space context. For example, they can generate test inputs only for batch programs that can be compiled with the LLVM infrastructure [3].

The FAQAS activity addresses the problems above. It is a joint work between the SnT Centre of the University of Luxembourg<sup>1</sup>, which is the prime, Gomspace Luxembourg<sup>2</sup> (GSL) and OHB Luxspace<sup>3</sup> (LXS). FAQAS led to the development of a toolset that addresses the challenges above. It includes four tools: *MASS* (Mutation Analysis for Space Software), *DAMAt* (DATA-driven Mutation Analysis with Tables), *SEMuS* (Symbolic Execution-based MUTant analysis for Space software), and *DAMTE* (DATA-driven Mutation TEsting). FAQAS last 24 months, with a budget of 500k Euro (360k to SnT, 70k to LXS, 70k to GSL).

Figure 1 provides an overview of the input and outputs of the FAQAS toolset. It relies on the idea of generating multiple modified versions of the software system under test (SUT), some are derived by modifying the implementation of the software (code-driven mutants) other by integrating a mutation API that alters the messages exchanged by the software components of the SUT (data-driven mutants). The SUT test suite shall be executed with all the mutants, if it is effective then it shall fail with each of them. The mutants for which a failure is not observed are said to be *live* and indicate a pitfall in the test suite. All the FAQAS tools take as input the software under test (SUT), its test suite, and a set of configuration files.

*MASS* generates code-driven mutants. It integrates a pipeline of solutions that make mutation analysis feasible with large SUT. The three main contributions of *MASS* are (1) the automated identification of trivially equivalent mutants using an ensemble of compiler optimization options, (2) the computation of the mutation score based on mutant sampling with fixed size confidence interval approach (FSCI), (3) the automated identification of equivalent mutants based on coverage. *MASS* reports the set of live mutants, the set of killed mutants (i.e., mutants that are discovered by

<sup>1</sup><https://www.en.uni.lu/snt>

<sup>2</sup><https://gomspace.com/>

<sup>3</sup><https://luxspace.lu/>

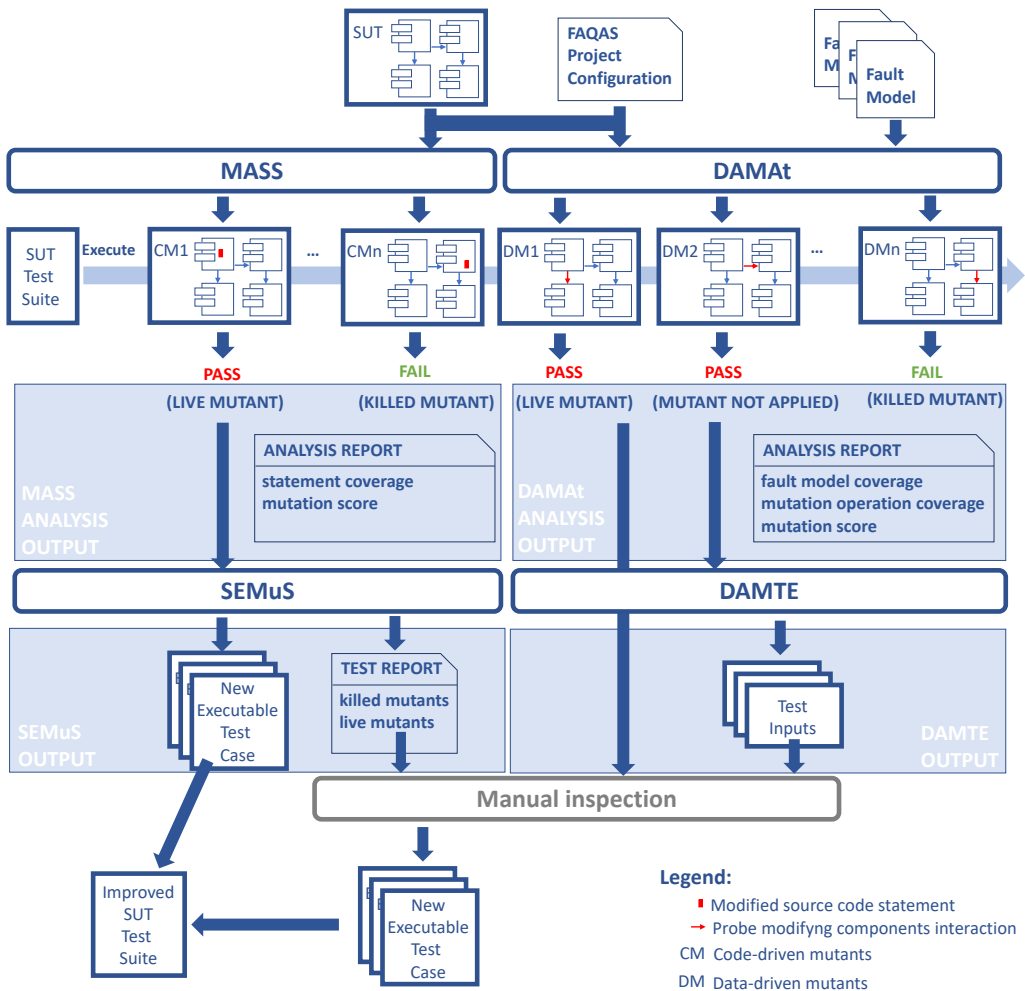


Fig. 1. Overview of the FAQAS toolset

the test suite), and information useful to draft a verification report, which includes the statement coverage of the SUT test suite and the mutation score (i.e., the percentage of mutants discovered by the test suite).

*DAMAt* generates mutants for data-driven mutation analysis. Data-driven mutation analysis is a research contribution of FAQAS. Instead of mutating the implementation of the SUT, it consists of altering the data exchanged by software components. *DAMAt* relies on fault models that specify how to mutate the data exchanged by software components through data-driven mutation operators. *DAMAt* can automatically alter data that is stored in data buffers (e.g., before serialization on the communication channel). *DAMAt* enables the simulation of faults that affect simulated components (e.g., sensors), which is not feasible with traditional, code-driven mutation analysis. *DAMAt* generates as output a set of killed mutants (i.e., mutants that, during testing, successfully alter the data, and lead to test case failures), a set of live mutants (i.e., mutants that, during testing, successfully alter the data, but do not lead to test case failures), and a set of mutants not applied

(i.e., mutants that, during testing, could not alter any data because the data they target is never exercised by the SUT); also, it provides information useful to draft a verification report, which includes the fault model coverage (i.e., percentage of fault models with at least one mutant applied), the mutation operation coverage (i.e., percentage of mutants applied), and the mutation score.

SEMUS automatically generates executable unit test cases based on code-driven mutation analysis results. The generated unit test cases detect mutants not detected by the original test suite. The generated test cases include test oracles that shall be manually validated by engineers, which enables detecting faults. The generated test cases can be integrated into regression test suites.

SEMUS takes as input the list of live mutants detected by MASS. It generates a set of additional test cases that can be integrated into the SUT test suite. Also, it reports the list of killed mutants and the list of mutants that remain live (i.e., for which SEMUS did not generate a test case that kill them). Live mutants shall be manually inspected by engineers to either determine if they are equivalent or to manually derive a test case capable of killing them.

DAMTE is a manual procedure supported by an automated symbolic execution toolset; it automatically identifies the test inputs that make software components exchange the data targeted by data-driven mutation operators. The derived test inputs can then be manually integrated into the SUT test suite.

## 1.1 Objectives

The FAQAS activity had the following two main objectives:

- O1 Searching for an alternative or complementary method of measuring the effectiveness of a test suite (i.e., test-suite verification).
- O2 Searching for an alternative or complementary method to build a test suite (i.e., test cases generation).

Concerning O1, FAQAS led to the development of tools (i.e., MASS and DAMAT) that enable the assessment of test suites by simulating different types of faults (implementation errors and high-level integration problems). Empirical evaluation with software provided by project partners and ESA has shown that the toolset enables detecting relevant test suite limitations (i.e., lack of assertions to verify results, relevant inputs not being tested). Also, it showed that the inspection of mutants enable detecting bugs that affect the software. Finally, preliminary projections made by consortium partners show that the cost of test suite assessment using the FAQAS tools (i.e., engineers' effort) are justified by the highly-valuable benefits (e.g., detection of test suite and software limitations during early development stages, potential avoidance of failures in deployed software).

Concerning O2, FAQAS has shown that tools based on symbolic execution (i.e., SEMUS and DaMTE) can automatically generate test cases that detect mutants not detected by the test suite under analysis. However, the research performed in FAQAS has shown that symbolic execution tools are affected by a number of limitations (e.g., dealing with floating point instructions and external components) that limit the feasibility of fully automated test generation. Indeed, only unit test cases not involving complex floating point operations can be automatically generated with state-of-the-art solutions including the FAQAS toolset.

Below, we report on how FAQAS contributed to address the **detailed objectives** of the activity:

- **Detailed Objective 1:** *To perform a comprehensive analysis and survey of mutation testing.*  
FAQAS has delivered a comprehensive survey of the software engineering literature on mutation testing with the following main findings:
  - The literature on mutation analysis/testing mostly focuses on modifying the code of the software under test (hereafter, code-driven approaches). Some approaches rely on

- modifying models, but they aim to generate test cases not assessing test suites. There are no approaches that assess test suites by changing the data generated by software components (hereafter, data-driven approaches).
- The mutation operators widely adopted to perform code-driven mutation analysis are the sufficient set of operators and the set of deletion operators. Other operators did not receive the same degree of attention in the literature. For example, higher-order mutation operators are reported to be easier to kill than the first-order ones (i.e., they are less effective in assessing test suites limitations); consequently, they had been adopted less in empirical studies.
  - The literature lacks mutation analysis approaches that enable simulating errors in the presence of simulated components (e.g., sensors).
  - Scalable approaches targeting mutation testing (i.e., automatically generating test cases that kill mutants) are few. The most promising ones rely on symbolic execution based on LLVM, which might be inapplicable for onboard flight software compiled for specific architectures.
  - **Detailed Objective 2:** *To prototype the mutation testing process to be applied on space software.* FAQAS has conducted a detailed analysis and preliminary experiments to select the existing mutation analysis, fault injection, fuzzing, and test generation tools that demonstrated to be reusable for the definition of an automated mutation testing toolset. Some results of the FAQAS evaluation are reported in the following.
    - Most existing mutation analysis tools rely on the mutation of LLVM [20] bitcode, which is often infeasible with space software projects.
    - The most advanced data mutation tool is the Peach fuzzer [23]; however, since it has been developed for other purposes (e.g., fuzzing web applications or desktop utilities) it presents an architecture that can hardly be adapted to mutate data in real-time space software.
    - The automated selection of test inputs that can kill mutants might be supported by either bounded model checking (BMC) [19] or symbolic execution [1]; however, existing symbolic execution tools (e.g., KLEE-SEMu [2]) are more stable and professional than BMC ones.
    - Finally, complex test inputs (e.g., sequences of hierarchical structures) might be generated by model-based approaches; however, advanced academic tools remain in a prototype state.
    - The analysis led to the identification of SRCIrror [17] and KLEE/SEMu as tools that might be extended to build the code-driven components of the FAQAS toolset. Data-driven approaches, instead, need to be built from scratch.
  - **Detailed Objective 6:** *To define and develop the mutation testing toolset supporting this methodology.* The FAQAS project has led to a toolset that includes **MASS** (Mutation Analysis for Space Software, TRL 5), **DAMAt** (Data-driven Mutation Analysis with Tables, TRL 4), **SE-MuS** (Symbolic Execution-based MUTant analysis for Space software, TRL 3), and **DAMTE** (Data-driven Mutation TEsting, TRL 2). Details are provided in Section 2.
  - **Detailed Objective 3:** *To empirically evaluate mutation testing by applying it to space software use cases.* ESA, LXS, and GSL have provided case study subjects that include utility and networking libraries, support tools, and whole systems. They come with test suites of different nature (unit, integration, and system-level). The heterogeneity of case studies and test suites enabled the evaluation of the mutation analysis and testing approaches in a range of representative scenarios. Details are provided in Section 3.
  - **Detailed Objective 4:** *To evaluate the applicability, scalability, efficiency and effectiveness of the approach in the space domain; and to identify limitations of the approach.* The project conducted an extensive evaluation of MASS, DAMAT, and SEMuS; also, it conducted a preliminary feasibility study for DAMTE. Our results (details in Section 4) show that

- MASS enables code-driven mutation analysis in the space context. The most effective solutions to improve scalability and mutation score accuracy are mutants sampling and equivalence metrics based on compiler optimizations, respectively. To guarantee a scalable mutation testing process and the accurate computation of the mutation score, mutants sampling should be based on sequential analysis relying on fixed-width sequential confidence interval, a research discovery done within FAQAS.
- DAMAt enables an efficient detection of relevant test suite shortcomings with less mutants to be inspected than MASS. It enabled the detection of a range of limitations including message types not being exchanged between software components, input partitions not being tested, imprecise oracles.
- SEMuS shows that symbolic execution can be successfully used to select test inputs that kill live mutants within unit test cases. However, unsurprisingly, it cannot be adopted when it is necessary to rely on external components (e.g., networks or simulators), in such cases, which are common for integration and system test suites, symbolic execution alone is insufficient to generate test cases.
- DAMTE is largely affected by the problems affecting SEMuS with the consequence that it requires large manual effort to be used.
- **Detailed Objective 5:** *To evaluate how mutation testing can be integrated into a typical verification & validation life cycle of space software, and to define the mutation testing methodology.* FAQAS has provided preliminary ideas about the definition of guidelines for the adoption of mutation analysis and testing strategies within ECSS activities (see Section 6). The proposed guidelines support both quality assurance activities described in ECSS standards [11, 12] and Independent Software Verification and Validation (ISVV) practices [13]. Finally, FAQAS has delivered a method for the assessment and improvement of software based on the results produced by the FAQAS toolset.
- **Detailed Objective 7:** *To foster the use of this methodology among the different software and independent software verification and validation suppliers.* The FAQAS toolset has been delivered to the project industry partners (i.e., GomSpace and LuxSpace), who validated it (see Section 5). Also, the description and empirical results concerning some of the tools part of the FAQAS toolset have been described in papers submitted to top academic venues in the field of software engineering. A paper concerning MASS has already been published in IEEE Transactions on Software Engineering [7]. Adoption among practitioners is a longer term objective that will be targeted during the maintenance phase of the project.

## 1.2 Outputs

The activity lead to the following tangible outputs:

- Tool: MASS. It reaches TRL 5: it is a configurable tool, with a user manual, that can be applied to whole software systems to perform mutation analysis. It has been installed on third party premises (i.e., GSL and LXS development environment) and independently used by third-party engineers (i.e., GSL and LXS) on relevant cases (e.g., projects not shared with SnT).
- Tool: DAMAt. It reaches TRL 4: it is a configurable tool, with a user manual, that can be applied to whole software systems to perform mutation analysis. It has been installed on third party premises (i.e., GSL and LXS development environment) and independently used by third-party engineers (i.e., GSL and LXS) on the case study subjects of the project.
- Tool: SEMuS. It reaches TRL 3: it is a configurable tool, with a user manual, that can be applied to a subset of source files for space software systems to perform test generation. It

has been installed on third party premises (i.e., GSL and LXS development environment) and independently used by third-party engineers (i.e., GSL and LXS) on a subset of source files belonging to the case study subjects of the project.

- Tool: DAMTE. It reaches TRL 2: it is provided as an extension of DAMAt, manual effort and scaffolding is needed to apply it to new projects. It has been applied to one case study subject of the project.
- E40C/Q80C documentation for the FAQAS toolset; it includes SVS, SVaIR, SUTR, SUTP, SUM, SSS, SRF, SReID, SPAP, SDD, SCF, IRD.
- Demonstration videos for MASS, DAMAt, and SEMuS.
- Research paper about MASS methodology published in IEEE Transactions on Software Engineering[7].
- Research paper about DAMAt submitted to ICSE'22.
- Research paper about MASS tool submitted to ICSE'22.

## 2 FAQAS METHODOLOGY

In the following, we describe how the results generated by the FAQAS toolset enable the assessment and improvement of a test suite.

### 2.1 Code-driven Mutation Analysis: MASS

Figure 2 provides an overview of MASS. It consists of ten steps described below.

**2.1.1 Step 0: Configure MASS.** Step 0 concerns the configuration of our toolset. The main configuration choices to be made by the engineer before running mutation analysis are: *Selecting the source files to mutate* (generally, all the source files of the SUT shall be considered for mutation); *Selecting the sampling strategy* (if the test suite of the SUT takes more than one hour to be executed, we suggest to rely on the *FSCI* mutant sampling strategy, otherwise, engineers can execute all the mutants); *Enabling test suite reduction and prioritization* (this choice enables MASS to further reduce test execution time by executing only a portion of the selected test cases based on statement coverage).

**2.1.2 Step 1: Collect SUT Test Data.** In Step 1, the test suite is executed against the SUT and code coverage information is collected. More precisely, we rely on the combination of gcov [6] and GDB [15], enabling the collection of coverage information for embedded systems without a file system [28].

**2.1.3 Step 2: Create Mutants.** In Step 2, we automatically generate mutants for the SUT by relying on a set of selected mutation operators, which are listed in Table 1.

**2.1.4 Step 3: Compile mutants.** In Step 3, we compile mutants by relying on an optimized compilation procedure that leverages the build system of the SUT. To this end, we have developed a toolset that, for each mutated source file: (1) backs-up the original source file, (2) renames the mutated source file as the original source file, (3) runs the build system (e.g., executes the command `make`), (4) copies the generated executable mutant in a dedicated folder, (5) restores the original source file.

**2.1.5 Step 4: Remove equivalent and redundant mutants based on compiled code.** In Step 4, we rely on trivial compiler optimizations to identify and remove equivalent and redundant mutants. We compile the original software and every mutant multiple times once for each every available optimization option (i.e., `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast` in GCC) or a subset of them. The outcome of Step 4 is a set of *unique mutants*, i.e., mutants with compiled code that differs from the original software and any other mutant.

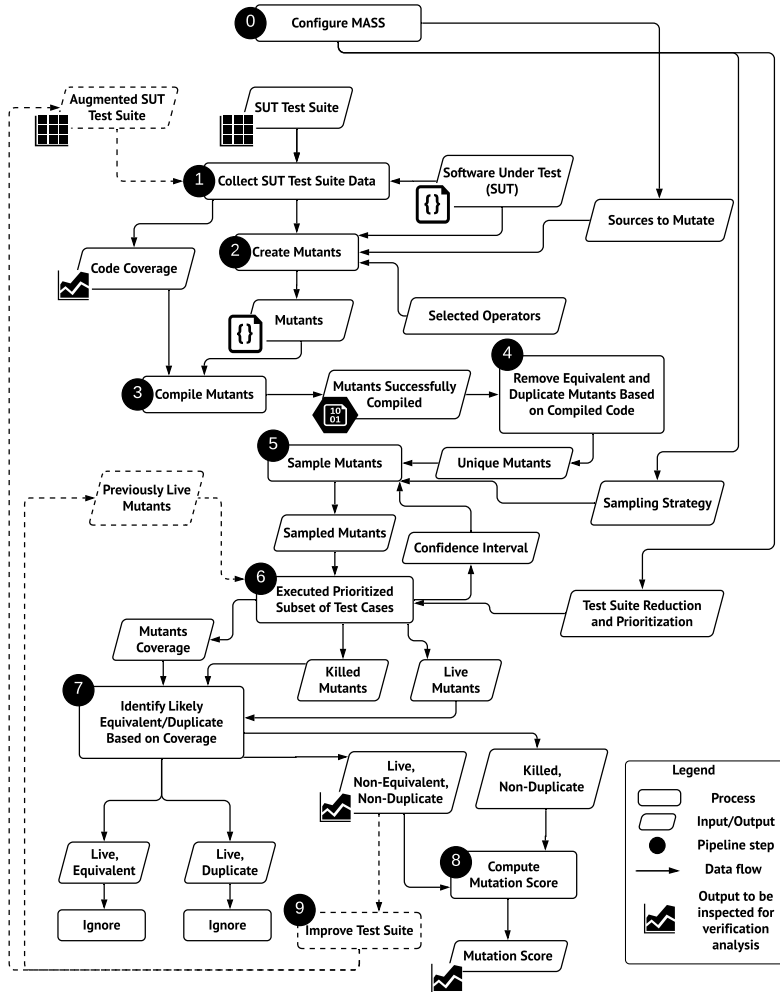


Fig. 2. Overview of the MASS workflow

**2.1.6 Step 5: Sample Mutants.** In Step 5, MASS samples the mutants to be executed to compute the mutation score. Our pipeline supports different sampling strategies: *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*. The strategies *proportional uniform sampling* and *proportional method-based sampling* were selected based on the results of Zhang et al. [29], who compared eight strategies for sampling mutants. The *uniform fixed-size sampling* strategy stems from the work of Gopinath et al. [16] and consists of selecting a fixed number  $N_M$  of mutants for the computation of the mutation score. We introduced the *uniform FSCI sampling* strategy that determines the sample size dynamically, while exercising mutants, based on a fixed-width sequential confidence interval approach. With *uniform FSCI sampling*, we introduce a cycle between Step 6 and Step 5, such that a new mutant is sampled only if deemed necessary. More precisely, MASS iteratively selects a random mutant from the set



Table 1. Implemented set of mutation operators.

	Operator	Description*
Sufficient Set	ABS	$\{(v, -v)\}$
	AOR	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, *, /, \%\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, *=, /=, \%=\} \wedge op_1 \neq op_2\}$
	ICR	$\{(i, x) \mid x \in \{1, -1, 0, i + 1, i - 1, -i\}\}$
	LCR	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, \ \} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&=,  =, \&=\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&,  , \&\&\} \wedge op_1 \neq op_2\}$
	ROR	$\{(op_1, op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\}\}$ $\{(e, !(e)) \mid e \in \{\text{if}(e), \text{while}(e)\}\}$
	SDL	$\{(s, \text{remove}(s))\}$
	UOI	$\{(v, -v), (v, v-), (v, ++v), (v, v++)\}$
	OODL	AOD
LOD		$\{((t_1 \text{ op } t_2), t_1), ((t_1 \text{ op } t_2), t_2) \mid op \in \{\&\&, \ \}\}$
ROD		$\{((t_1 \text{ op } t_2), t_1), ((t_1 \text{ op } t_2), t_2) \mid op \in \{>, >=, <, <=, ==, !=\}\}$
BOD		$\{((t_1 \text{ op } t_2), t_1), ((t_1 \text{ op } t_2), t_2) \mid op \in \{\&,  , \wedge\}\}$
SOD		$\{((t_1 \text{ op } t_2), t_1), ((t_1 \text{ op } t_2), t_2) \mid op \in \{>, <\}\}$
Other	LVR	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0), (true, false), (false, true)\}\}$

\*Each pair in parenthesis shows how a program element is modified by the mutation operator on the left; we follow standard syntax [18]. Program elements are literals ( $l$ ), integer literals ( $i$ ), boolean expressions ( $e$ ), operators ( $op$ ), statements ( $s$ ), variables ( $v$ ), and terms ( $t_i$ , which might be either variables or literals).

of unique mutants and exercises it using the SUT test suite. The result of each mutant execution (i.e., killed or live) is treated as a Bernoulli trial that is used to compute the confidence interval according to the FSCI method. To compute the confidence interval for the FSCI analysis, we rely on the Clopper-Pearson method since it is reported to provide the best results [5].

**2.1.7 Step 6: Execute prioritized subset of test cases.** In Step 6, we execute a prioritized subset of test cases. We select only the test cases that satisfy the reachability condition (i.e., cover the mutated statement) and execute them in sequence. To determine how dissimilar two test cases are and, consequently, how likely they exercise the mutated statement with different values, we rely on Cosine similarity.

**2.1.8 Step 7: Discard Mutants.** In this step, we identify likely nonequivalent mutants by relying on code coverage information collected in the previous step. A mutant is considered nonequivalent when the distance from the original program is non null, for at least one test case.

**2.1.9 Step 8: Compute Mutation Score and Analysis Output.** The *mutation score* (MS) is computed as the percentage of killed nonduplicate mutants (hereafter, *KND*) over the number of nonequivalent, nonduplicate mutants identified in Step 7):

$$MS = \frac{|KND|}{|LNEND| + |KND|} \quad (1)$$

The main output of MASS is a file named *MASS\_RESULTS*. An example of the *MASS\_RESULTS* report is presented in Listing 1. Within file *MASS\_RESULTS*, the *first metric* to be inspected is the *Statement coverage* (i.e., the percentage of statements being covered). Since MASS generates mutants only for the statements being exercised by the test suite, a high mutation score in the presence of a low statement coverage cannot indicate that the test suite has high quality.

The *second metric* to be inspected is the *MASS mutation score*. It provides an indication of the quality of the test suite based on mutation analysis results. According to the literature on the topic, achieving a high mutation score improves significantly the fault detection capability of a test suite [25]; also, a very high mutation score (i.e., above 0.75) ensures a higher fault detection rate than the one obtained with other coverage criteria, such as statement and branch coverage [4].

```

1 ##### MASS Output #####
2 ## Total mutants generated: 28071
3 ## Total mutants filtered by TCE: 6918
4 ## Sampling type: fsci
5 ## Total mutants analyzed: 461
6 ## Total killed mutants: 369
7 ## Total live mutants: 92
8 ## Total likely equivalent mutants: 53
9 ## MASS mutation score (%): 90.44
10 ## List A of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_a
11 ## List B of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_b
12 ## Number of statements covered: 1973
13 ## Statement coverage (%): 100
14 ## Minimum lines covered per source file: 2
15 ## Maximum lines covered per source file: 138

```

Listing 1. MASS output obtained with the MLFS case study subject.

Three additional relevant output files generated by MASS are *filtered\_live*, *useful\_list\_a* and *useful\_list\_b*. They contain the names of the live mutants. The file *useful\_list\_a* provides a list of mutants that are likely non redundant with each other because when tested by the SUT test suite they lead to a statement coverage profile (i.e., the set of statements covered during their execution) that differs. The file *useful\_list\_b* provides a list of mutants that are likely redundant with the ones appearing in the file *useful\_list\_a*. The mutants within file *useful\_list\_a* are sorted according to their diversity (i.e., the mutants on top are likely very different from each other). The file *filtered\_live* is the union of the mutants appearing in the files *useful\_list\_a* and *useful\_list\_b*.

**2.1.10 Step 9: Improve Test Suite.** Step 9 can be performed manually or can be automated through SEMuS. It consists of deriving test inputs that kill live mutants.

To manually perform Step 9, engineers shall inspect all the mutants appearing in the file *useful\_list\_a*. For each mutant, the engineer shall implement a test case capable of killing the mutant (i.e., a test case that fails with the mutant but not with the original software). In general, since a same test case may kill more than one mutant, we suggest to derive test inputs for a subset of the mutants in *useful\_list\_a* and then rerun the mutation analysis process. When rerunning the mutation analysis process, engineers shall focus the mutation analysis on the mutants appearing in *useful\_list\_a* and in *useful\_list\_b*. This is done by re-executing mutation analysis from Step 6 (*Execute mutants*).

When automated test generation with SEMuS is feasible; we suggest to rely on SEMuS to automatically generate test cases for all the mutants appearing in *useful\_list\_a* and in *useful\_list\_b* (see Section 2.2).

When identifying inputs that kill mutants (either manually or with SEMuS) engineers may detect equivalent mutants. Equivalent mutants shall be removed from the list of mutants considered for the analysis.

If mutation analysis has been performed through mutants sampling (e.g., with *FSCI*), after test suite improvement (i.e., after introducing test cases that kill all the mutants in *useful\_list\_a* and in *useful\_list\_b*), it is necessary to re-run mutation analysis to estimate the mutation score for the whole system.

## 2.2 Code-driven Mutation Testing: SEMuS

Figure 3 provides the workflow of SEMuS. The list of live mutants processed by SEMuS coincides with the list of mutants appearing in the file *filtered\_live* presented in Section 2.1.

SEMuS consists of five components, which are *Test Template Generator*, *Pre-SEMu*, *KLEE-SEMu*, *KTest to Unit Test*, and *LLVM*.

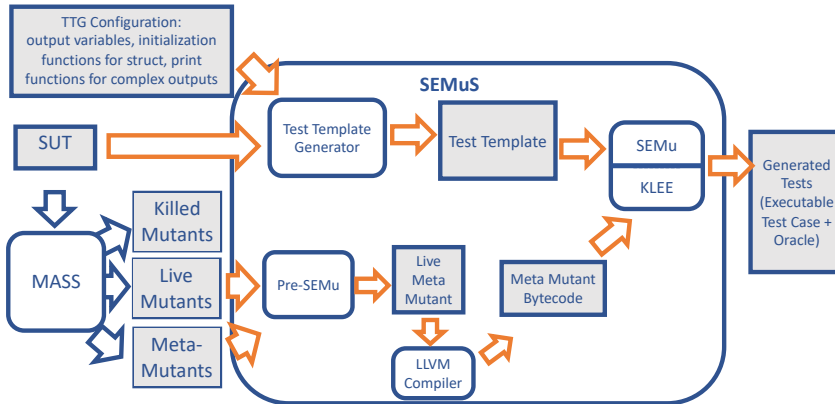


Fig. 3. FAQAS-SEMUS Architecture and Workflow

The *Test Template Generator* (TTG) component automates the generation of templates for the symbolic execution search. The component receives as inputs the SUT source code and the list of SUT functions. Listing 2 shows an example of a test template generated by the TTG. The TTG generates a template for every SUT function. The TTG parses the function arguments and declares them symbolic through use of the KLEE function `klee_make_symbolic`. Then, it adds a call to the function under analysis with symbolic values, and it saves the return value into a support variable (i.e., `result_faqs_semu` in Listing 2). Finally, it generates a number of invocations of the *printf* function that print the value of the software outputs and adds a return statement with the value returned by the function under test (e.g., `result_faqs_semu` in Listing 2).

```

1 int main(int argc, char** argv) {
2     // Declare variable to hold function returned value
3     _Bool result_faqs_semu;
4     // Declare arguments and make input ones symbolic
5     unsigned long pVal;
6     int pErrCode;
7     klee_make_symbolic(&pVal, sizeof(pVal), "pVal");
8     // Call function under test
9     result_faqs_semu = T_INT_IsConstraintValid(&pVal, &pErrCode);
10    // Make some output
11    printf("FAQAS-SEMU-TEST_OUTPUT: %d\n", pErrCode);
12    printf("FAQAS-SEMU-TEST_OUTPUT: %d\n", result_faqs_semu);
13    return (int)result_faqs_semu;
14 }
  
```

Listing 2. SEMuS test template.

The *Pre-SEMu* component includes and compiles all the live mutants (i.e., MASS output) into a single bytecode file named the *Meta Mutant*. SEMu will select which mutant to consider for test

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #include "asn1crt.c"
5 #include "asn1crt_encoding.c"
6 #include "asn1crt_encoding_uper.c"
7
8
9 int main(int argc, char** argv)
10 {
11     (void)argc;
12     (void)argv;
13
14     // Declare variable to hold function returned value
15     _Bool result_faqs_semu;
16
17     // Declare arguments and make input ones symbolic
18     unsigned long pVal;
19     int pErrCode;
20     memset(&pVal, 0, sizeof(pVal));
21     const unsigned char pVal_faqs_semu_test_data[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
22     x00, 0x00};
23     memcpy(&pVal, pVal_faqs_semu_test_data, sizeof(pVal)); // Unsigned val is 0
24
25     // Call function under test
26     result_faqs_semu = T_INT_IsConstraintValid(&pVal, &pErrCode);
27
28     // Make some output
29     printf("FAQS-SEMU-TEST_OUTPUT: pErrCode = %d\n", pErrCode);
30     printf("FAQS-SEMU-TEST_OUTPUT: result_faqs_semu = %d\n", result_faqs_semu);
31     return (int)result_faqs_semu;
32 }

```

Listing 3. Generated test case

generation based on a parameter. The compilation of the Meta Mutant into LLVM bitcode is enabled by the *LLVM* compiler infrastructure.

*KLEE-SEMu* is the underlying test generation component. This component receives as inputs the *LLVM bitcode* of the *Meta Mutant* and the *Test Template* for the function under test, and proceeds to apply dynamic symbolic execution to generate test inputs to kill the mutants. The output of this component are the *KLEE tests*. A *KLEE test* is a binary file that contains information about the execution of *KLEE* such as the entry point of the analysis, and the generated test inputs.

The component *KTest to Unit Test (KTU)* converts a *KLEE test* into a human readable, compilable, and executable C test case. The unit test case generated by *KTU* matches the test template generated by *TTG* except for the declaration of variables where symbolic variables are replaced with concrete variables initialized with the values stored in the *KTest* file. Listing 3 shows an example of a test case generated for a mutant present in the function `T_INT_IsConstraintValid`.

The mutants for which *SEMuS* does not generate a test case<sup>4</sup> shall be manually inspected by engineers to determine if they are equivalent to the original software. The test cases generated by *SEMuS* can instead be integrated as a regression test suite.

The main output of *SEMuS* is a report file named *AnalysisReport.csv*, which includes the number of mutants that were killed by *SEMuS* (i.e., the tool has generated an input that kills the mutant), the number of mutants that were not killed by *SEMuS* (i.e., the tool could not generate an input to kill a mutant), and a list showing the status of each mutant (i.e., killed or live).

<sup>4</sup>In our toolset, such mutants are identified by looking for empty folders within the output folder `direct/TEMPLATE/FAQAS_SEMu-out/produced-unittests`.

Table 2. Data-driven mutation operators

Fault Class	Description
Value above threshold (VAT)	Replaces the current value with a value above the threshold $T$ for a delta $\Delta$ .
Value below threshold (VBT)	Replaces the current value with a value below the threshold $T$ for a delta $\Delta$ .
Value out of range (VOR)	Replaces the current value with a value out of the range $[MIN; MAX]$ .
Bit flip (BF)	A number of bits randomly chosen in the positions between $MIN$ and $MAX$ are flipped.
Invalid numeric value (INV)	Replace the current value with a mutated value that is legal (i.e., in the specified range) but different than current value.
Illegal Value (IV)	Replace the current value with a value that is equal to the parameter <i>VALUE</i> .
Anomalous Signal Amplitude (ASA)	The mutated value is derived by amplifying the observed value by a factor $V$ and by adding/removing a constant value $\Delta$ from it.
Signal Shift (SS)	The mutated value is derived by adding a value $\Delta$ to the observed value.
Hold Value (HV)	This operator keeps repeating an observed value for $V$ times. It emulates a constant signal replacing a signal supposed to vary.
Fix value above threshold (FVAT)	In the presence of a value above the threshold, it replaces the current value with a value below the threshold $T$ for a delta $\Delta$ .
Fix value below threshold (FVBT)	It is the counterpart of FVAT for the operator VBT.
Fix value out of range (FVOR)	In the presence of a value out of the range $[MIN; MAX]$ it replaces the current value with a random value within the range.

### 2.3 Data-driven Mutation Analysis: DAMAt

Data-driven mutation analysis aims to evaluate the effectiveness of a test suite in detecting *semantic interoperability faults*. It is achieved by modifying (i.e., mutating) the data exchanged by CPS components. It generates *mutated data* that is representative of data that might be observed at runtime in the presence of a component that behaves differently than expected in the test case; also, it mutates data that is not automatically corrected by the software (e.g., through cyclic redundancy check codes) and thus causes software failures (i.e., the mutated data shall have a different semantic than the original data). For these reasons, data mutation is driven by a fault model specified by the engineers based on domain knowledge.

The *DAMAt* fault model is a tabular block model. It enables the modelling of data that is exchanged through a specific data structure: the data buffer. This was decided because it is a simple and widely adopted data structure for data exchanges between components in CPS. The *DAMAt* fault model enables the specification of the format of the data exchanged between components along with the type of faults that may affect such data. We refer to the data exchanged by two components as *message*. For a single CPS, more than one fault model can be specified (e.g., one for each message type). The *DAMAt* fault model enables engineers to specify (1) the *position* of each data item in the buffer, (2) their *span*, and (3) their *representation type*. Further, for each data item, *DAMAt* enables engineers to specify one or more data faults using the mutation operator identifiers. For each operator, the engineer shall provide values for the required configuration parameters. Table 2 provides the list of mutation operators included in *DAMAt* along with their description.

The *DAMAt* mutation operators generate *mutated data item instances* through one or more *mutation procedures*, which are the functions that generate a mutated data item instance given a correct data item instance observed at runtime. For example, the *VAT* operator includes only one mutation procedure (i.e., setting the current value above the threshold) while the *VOR* operator includes two mutation procedures, which are (1) replacing the current value with a value above the specified valid range and (2) replacing the current value with a value below the valid range. The operators *VOR*, *BF*, *INV*, and *SS* have been inspired by related work [9, 22, 26]; the operators *VAT*, *VBT*, *FVAT*, *FVBT*, *FVOR*, *IV*, *ASA*, and *HV* are a contribution of *FAQAS*.

*DAMAt* works in six steps, which are shown in Figure 5. In Step 1, based on a methodology provided with the *DAMAt* documentation, the engineer prepares a fault model specification tailored to the SUT. Our methodology enables the specification of all possible interoperability problems in the SUT while minimizing equivalent and redundant mutants.

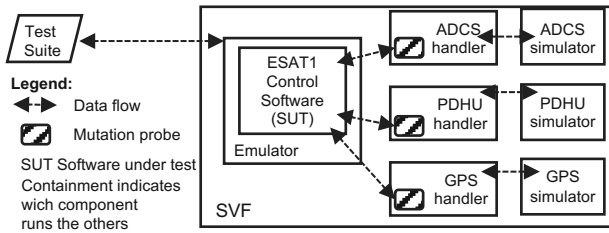


Fig. 4. Data mutation probes integrated into ESAIL.

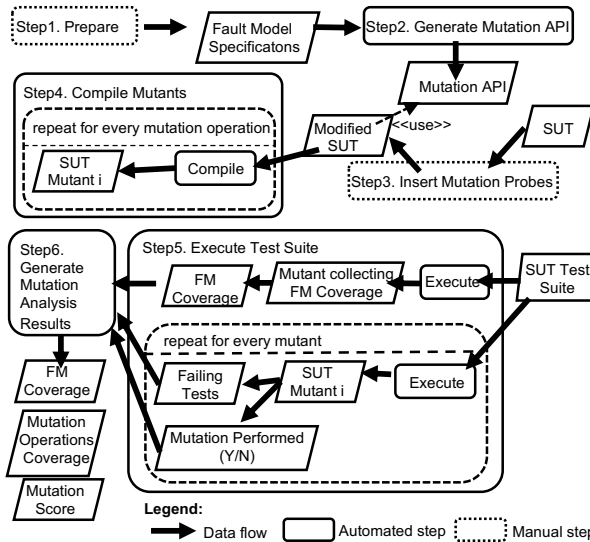


Fig. 5. The DAMAt process.

In Step 2, DAMAt generates a mutation API with the functions that modify the data according to the provided fault model. These functions select the data item to mutate and the mutation procedure to apply based on the mutant under test.

In Step 3, the engineer modifies the SUT by introducing mutation probes (i.e., invocations to the mutation API) into it. Instead of modifying the SUT the engineer may modify the test harness (e.g., the SVF simulator); such choice depends on the software under test, if the test cases are executed through a simulator, such choice prevents introducing damaging changes into the SUT (e.g., delay task execution and break strict real-time requirements). The effort required by the engineer is minimal; indeed, the exchange of data between components is usually managed in a single location (e.g. the function that serializes the data buffer on the network) and thus it is usually sufficient to introduce one function call for each message type to mutate.

In Step 4, DAMAt generates and compiles mutants. Since the DAMAt mutation operators may generate mutated data by applying multiple mutation procedures, DAMAt may generate several mutants, one for each data mutation operation (i.e., a mutation procedure configured for a data item). The mutant generation is invisible to the end-user who does not need to modify the source code further.

In Step 5, *DAMAt* executes the test suite with all the mutants including a mutant (i.e., the coverage mutant) which does not modify the data but traces the coverage of the fault model. The information collected by the coverage mutant enables the execution, for every mutant, of the subset of test cases that cover the message type targeted by the mutant, thus speeding up mutation analysis.

In Step 6, *DAMAt* generates mutation analysis results: *fault model coverage*, *mutation operation coverage*, and *mutation score*. These metrics measure the frequency of the following scenarios: (case 1) the message type targeted by a mutant is never exercised, (case 2) the message type is covered by the test suite but it is not possible to perform some of the mutation operations (e.g., because the test suite does not exercise out-of-range cases), (case 3) the mutation is performed but the test suite does not fail.

*Fault model coverage (FMC)* is the percentage of fault models covered by the test suite. Since we define a fault model for every message type exchanged by two components, it provides information about the extent to which the message types actually exchanged by the SUT are exercised and verified by the test suites.

*Mutation operation coverage (MOC)* is the percentage of data items that have been mutated at least once, considering only those that belong to the data buffers covered by the test suite. It provides information about the input partitions covered for each data item.

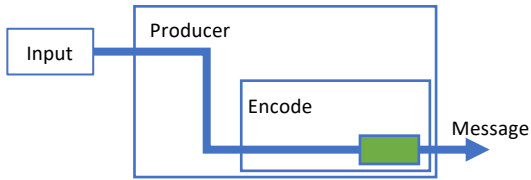
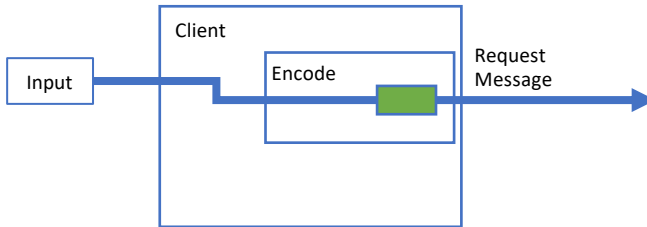
The *mutation score (MS)* is the percentage of mutants killed by the test suite (i.e., leading to at least one test case failure) among the mutants that target a fault model and for which at least one mutation operation was successfully performed. It provides information about the quality of test oracles; indeed, a mutant that performs a mutation operation and is not killed (i.e., is *live*) indicates that the test suite cannot detect the effect of the mutation (e.g., the presence of warnings in logs). Also, a low mutation score may indicate missing test input sequences. Indeed, live mutants may be due to either software faults (e.g., the SUT does not provide the correct output for the mutated data item instance) or the software not being in the required state (e.g., input partitions for data items are covered when the software is paused); in such cases, with appropriate input sequences, the test suite would have discovered the fault or brought the SUT into the required state. Both poor oracles and lack of inputs indicate flaws in the test case definition process (e.g., the stateful nature of the software was ignored).

Finally, *DAMAt* generates a file named *final\_mutants\_table.csv*, which contains a list of all generated mutants, the definition of the mutation operator that generated them and their status. It is used to determine how to improve the test suite; indeed, it specifies which anomalous values were not discovered by the test suite.

## 2.4 Data-driven Mutation Testing: DaMTE

*DAMTE* support engineers in identifying test inputs so that all the mutants are applied (i.e., to have fault model coverage and mutation operation coverage reach 100%). It is applicable to two common software architectures: the producer-consumer and client-server architecture (see Figure 6).

To generate the required inputs we rely on an *extended data mutation probe*. The extended data mutation probe invokes a version of the *DAMAt* data mutation API that, instead of performing mutation operations, includes reachability assertions that can be used to force KLEE to generate a test input that reaches the assertion. The extended version of the data mutation API includes macro commands that enable to specify which functionality to trigger, either data-driven mutation analysis or data-driven mutation testing. It enables test generation by introducing the reachability statement `assert(false)` that enforces KLEE to identify inputs that cover the statement. Specifically, the reachability statement makes KLEE to (1) cover that branch, (2) terminate the symbolic execution, and (3) to solve the path condition to look for concrete test inputs that cover the branch.

**Producer-consumer****Client-server**


 Extended FAQAS mutation probe

Fig. 6. Data-driven mutation testing for different architectures.

### 3 FAQAS CASE STUDIES

The FAQAS toolset has been applied to six case study systems: *ESAIL*, *LIBGCSP*, *LIBParam*, *LIBUTIL*, *MLFS*, *ASN1SCC*.

*ESAIL* is a microsatellite developed by LXS in a Public-Private-Partnership with ESA and ExactEarth. For our empirical evaluation, we considered the onboard control software of *ESAIL* (hereafter, simply *ESAIL-CSW*), which consists of 924 source files with a total size of 187,116 LOC.

*LIBGCSP*, *LIBParam*, and *LIBUTIL* are utility libraries developed by GSL. *LIBGCSP* is a network protocol library including low-level drivers (e.g., CAN, I2C). *LIBParam* is a light-weight parameter system designed for GSL satellite subsystems. *LIBUTIL* is a utility library providing cross-platform APIs for use in both embedded systems and Linux development environments.

The *Mathematical Library for Flight Software*<sup>5</sup> (*MLFS*) implements mathematical functions ready for qualification. In FAQAS, we considered the unit test suite of *MLFS* (it achieves branch and MC/DC coverage).

*ASN1SCC*<sup>6</sup> is an open source ASN.1 compiler that generates C/C++ and SPARK/Ada code suitable for space systems. Also, it produces a test suite for the generated code achieving statement coverage adequacy. For our experiments, we apply the FAQAS toolset to assess the automatically generated test suite by mutating the generated code.

<sup>5</sup><https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software>

<sup>6</sup><https://github.com/ttsiodras/asn1scc>



For the validation of each tool in the FAQAS toolset, we selected case study systems with characteristics compatible with the requirements of the tool under test. Table 3 provides the list of case studies along with an indication of the type of mutation analysis/testing (i.e., code-driven or data-driven) and the tools they are targeted for.

Table 3. Case studies for the FAQAS activity.

Partner	Case study	Code-driven		Data-driven	
		MASS	SEMUS	DAMAt	DaMTe
LXS	System Test Suite for ESAIL	Y	N	Y	N
LXS	Unit Test Suite for ESAIL	Y	Y	N	N
GSL	Unit Test Suite for libUtil	Y	Y	N	N
GSL	Integration Test Suite for libgscsp	Y	N	Y	N
GSL	System Test Suite for libparam	Y	N	Y	Y
ESA	MLFS mathematical library	Y	Y	N	N
ESA	ASN1 Compiler	Y	Y	N	N

## 4 EMPIRICAL EVALUATION

The FAQAS activity has been evaluated through an extended empirical evaluation; below we summarize our findings.

### 4.1 MASS

Both GSL and LXS have manually inspected a subset of the live mutants identified by MASS. The inspection enabled industry partners to identify relevant shortcomings in their test suites:

- 57% of the live mutants are due to missing inputs. Of particular relevance are exceptional cases not being exercised by the test suite, which shows that engineers are not able to determine all the unexpected execution conditions that the SUT shall take care of.
- 23% of the live mutants were due to missing oracles. Such result is particularly relevant because it indicates that although engineers believe to have tested a relevant scenario, the absence of an appropriate oracle prevents them from automatically detecting failures that might be observed when running the test cases.
- The few remaining live mutants had been reported as either equivalent or not relevant (e.g., because concerning third party software).
- One fault was detected. More precisely, the implementation of a test case that detects the mutant has shown that the SUT provides an erroneous result.

Finally, our results show that MASS helps addressing scalability problems to a significant extent by reducing mutation analysis time by more than 70% across subjects. In practice, for large software systems like *ESAIL*, such reduction can make mutation analysis practically feasible; indeed, with 100 HPC nodes available for computation, *MASS* can perform the mutation analysis of *ESAIL-CSW* in half a day. In contrast, a traditional mutation analysis approach would take more than 100 days, thus largely delaying the development and quality assurance processes. Also, we demonstrated that the FSCI sampling approach implemented by *MASS* leads to an accurate estimation of the mutation score.

### 4.2 SEMuS

The empirical evaluation demonstrated the scalability of *SEMUS* for the case study subjects in which it can be successfully applied (i.e., *ASN1CC*, *MLFS*, and *LIBUTIL*). Our results also demonstrated the usefulness of *SEMUS*. Indeed, *SEMUS* enabled the identification of two faults in our case studies.

Also, the generated test cases concerned inputs that are relevant (according to specifications) but not tested by the test suites.

### 4.3 DAMAt

The empirical evaluation of DAMAt has demonstrated the effectiveness of the approach. Indeed, LXS has indicated that 57% out of the overall amount of 102 test suite problems detected by DAMAt were spotting major limitations of the test suite. Also, GSL has confirmed that the approach enabled the detection of relevant test suite shortcomings. One possible limitation of the approach is that it may introduce slow-downs that lead to non-deterministic failures when the test suite exercises brief interaction scenarios in which most of the operations performed concern the encapsulation of data into the network.

Our results confirm that (1) uncovered fault models (i.e., low *FMC*) indicate lack of coverage for certain message types (*UMT*) and, in turn, the lack of coverage of a specific functionality (i.e., setting the pulse-width modulation in *ESAIL-ADCS*); (2) uncovered mutation operations (i.e., low *MOC*) highlight the lack of testing of input partitions (*UIP*); (3) live mutants (i.e., low *MS*) suggest poor oracle quality (*POQ*).

Based on our evaluation, we observed that live mutants can be killed by introducing oracles that (1) verify additional entries in the log files (39 instances for *ESAIL-ADCS*, 1 instance for *ESAIL-GPS*), (2) verify additional observable state variables (14 instances for *ESAIL-ADCS*, 4 instances for *LIBParam*), and (3) verify not only the presence of error messages but also their content (2 instances for *ESAIL-ADCS*). Such oracles may consist of additional assertions that verify data values already produced by the software under test (i.e., no modification of the SUT is needed).

### 4.4 DAMTE

*DAMTE* aims to address a task (i.e., test generation at system and integration level) that is particularly difficult to address with state-of-the-art technology (e.g., test generation toolsets based on symbolic execution). For this reason, FAQAS only concerned the evaluation of the feasibility of *DAMTE*.

We relied on *DAMTE* to generate inputs for the *LIBParam* client API functions. Such inputs enable the exchange of messages between the *LIBParam* client and the *LIBParam* server. Overall, we conclude that the *DAMTE* approach may be feasible; however, it requires some manual effort for the configuration and execution of test cases which may limit its usefulness. The first step towards its large scale applicability is the improvement of underlying test generation tools and compiler procedures, such changes will facilitate *DAMTE* application to large projects without the need for manually creating test template files with dependencies.

## 5 INDUSTRIAL VALIDATION SUMMARY

The developed toolset has demonstrated to be useful in industrial contexts. The common limitation cross the different tools is the usability; indeed, all the tools require relevant effort to be set-up (however, LXS has reported that if at least 6% of the reported problems spot major limitations the benefits surmount costs). The need for manual effort mostly depends on the lack of a common development environment for different case study subjects. The identification of a reference platform for software development in industry context may facilitate the adoption of the FAQAS toolset.

Other limitations that need further research effort to simplify the adoption of the FAQAS toolset are the prioritization of mutants to be inspected, the need for a solution to compile whole SUTs with LLVM, the need for a solution to enable test generation in the presence of floating point variables, the need for a working solution to enable test generation based on data-driven mutation analysis results.

Below we report verbatim the positive and negative comments provided in the validation deliverables of the project by our industry partners.

## 5.1 Overall Comments

### POSITIVE COMMENTS

- *Using the FAQAS toolset is for sure cost-effective when more than the 6% of the detected major problems of the testsuite could have determined a software error would have gone undetected.*
- *Both these fields of research are considered from LuxSpace worth to be explored in a possible extension of the FAQAS project: (\*) the capability of the toolset to decrease the time of analysis of the results (\*) the capability of the toolset to generate automatically additional/updated test cases that may patch the current failing testsuite.*

## 5.2 MASS

### POSITIVE COMMENTS

- *The SUM contains all necessary information in a well written style. This includes an explanation of the library structure and the purpose of contained files, a description of all configuration variables within those files and instructions for running the toolset. Each important file is provided with its own subsection, allowing the end-user to get a clear understanding of the framework configuration.*
- *The MASS toolset offers many configuration options to the end-user. All these configuration options and parameters are well described in the SUM.*
- *The MASS tool is effective at identifying potential defects that are unanticipated by our current test suites. It is also worth mentioning that closer examination of the code inspired by this approach did seem to reveal actual defects in the software that were previously unknown.*
- *The FAQAS team used Singularity as a container system, however it is also possible to create a Docker image that allows running MASS mutation tests in a docker container.*
- *Configuration files can be stored together with source code and mounted as volumes. In principle this makes it trivial to spawn a new instance (horizontal scaling).*
- *The evaluation showed that the MASS tool would be considered as a useful addition to GomSpace's testing processes.*
- *Applying this approach to other libraries maintained by GomSpace could allow managers to direct efforts towards improving test suites identified as lower quality. This would lead to an overall improvement in both test suite and code quality.*

### NEGATIVE COMMENTS

- *The evaluation did identify that many abbreviations and acronyms are used within the SUM, some of them without explanation in the document. An expansion of the list of abbreviations and acronyms is recommended to improve the overall user experience.*
- **Action taken:** To address the comment above, SnT has improved the SUM accordingly.
- *The large degree of possibility is challenging for a new user to learn and can be overwhelming for first use. GomSpace recommends providing an interactive script with default values to ease this process.*
- *The initial (first-time) configuration process should be simplified to reduce the steep learning curve*

- **Action taken:** To address the two comments above, SnT has modified the MASS installation step, so that some of the tool configurations are now provided with default values (e.g., use of trivial compiler optimisations, sampling technique used, prioritized test suite).

### 5.3 DAMAt

#### POSITIVE COMMENTS

- *The SUM contains all necessary information in a well written style. This includes an explanation of the library structure and the purpose of contained files, a description of all configuration variables within those files and instructions for running the toolset. Each important file is provided with its own subsection, allowing the end-user to get a clear understanding of the framework configuration.*
- *The DAMAt toolset offers many configuration options to the end-user. All these configuration options and parameters are well described in the SUM. The ease of configuring these parameters has increased compared to the previous MASS tool. For example: DAMAt\_FOLDER=\$(pwd) and using this variable further for configuration significantly saves time.*
- *Analysis of the surviving mutants shows the toolset does identify valid (potential) test cases that the test suites currently miss. This implies the presence of missing test cases, often in areas that can be considered as challenging edge cases that are difficult for a developer or dedicated software tester to anticipate, and in some cases poorly written test cases. Based on the results generated, the DAMAt tool is effective at identifying potential defects and missing test cases that are unanticipated by our current test suites.*
- *DAMAt can be containerized. The FAQAS team used Singularity as a container system, however it is also possible to create a Docker image that allows running DAMAt mutation tests in a docker container. Configuration files can be stored together with source code and mounted as volumes. In principle this makes it trivial to spawn a new instance (horizontal scaling).*
- *The evaluation showed that the DAMAt would be considered as a useful addition to GomSpace's testing processes.*
- *After checking the output report, GomSpace realised that the test suite of libparam does not address certain side effects of functions. This already demonstrates the ability of the toolset to identify improvements that would raise the quality of existing test suites.*
- *Applying this approach to other libraries maintained by GomSpace could allow managers to direct efforts towards improving test suites identified as lower quality (i.e., those with more surviving mutants) or to set certain gateway thresholds (i.e., a certain proportion of mutants must be caught before a test suite is considered high enough quality to proceed). This would lead to an overall improvement in both test suite and code quality.*

#### NEGATIVE COMMENTS

- *The evaluation did identify that some of the missing steps in the SUM were present in the readme of the project. Without which it was quite difficult to configure some of the steps, as there is no explanation in the document.*
- *Finally, the SUM doesn't have sufficient information about fault models and how they are important to the process, analysis, and results. Only an example is present in the SUM. It would be useful to have some information explaining fault models, their significance, and some background of its correlation to probes, testcases etc.*
- **Action taken:** To address the two comments above, SnT has improved the SUM accordingly.

- *To apply DAMAt, we need to manually inject probes into the code, which requires prior additional knowledge/understanding of inner working of the software under test (SUT). Moreover, the injected probes should always be used only for tests. In production, they shouldn't be present and there is necessity of automated process that handles probes management.*
- *There is manual intervention of adding lots of probes and building a fault model which can be a time-consuming process.*
- *The addition of probes as manual step would make it more difficult to scale especially when it is the case of microservices when there are large number of interacting microservices.*
- **Action taken:** To address the three comments above, SnT has introduced a feature to the DAMAt pipeline that enables the user to leave simple comments in the source code that will be substituted with the mutation probes during the procedure. The new feature also reinstates the unmodified code once the execution of DAMAt is concluded.

## 5.4 SEMUS

### POSITIVE COMMENTS

- *The SEMUS offers many configuration options to the end-user. All these configuration options and parameters are well described in the SUM. There are also scripts that are used for automatic generation of JSON files and Test templates.*
- *Analysis of the generated testcases for the mutants identified valid bug for timestamp.c as well as missing Test cases. This implies the presence of missing test cases, often in areas that can be considered as challenging edge cases that are difficult for a developer or dedicated software tester to anticipate and in some cases poorly written test cases.*
- *SEMUS can be containerized. The FAQAS team used Singularity as a container system, however it is also possible to create a Docker image that allows running SEMUS test generation in a docker container. Configuration files can be stored together with source code and mounted as volumes. In principle this makes it trivial to spawn a new instance (horizontal scaling).*
- *However, when it is fully executed it does help with identifying the missing test cases in test suite and as a side-effect points out to potential bugs.*
- *After checking the output report, GomSpace realised that the test suite of libutil does not address certain tests. This already demonstrates the ability of the SEMUS to identify improvements that would raise the quality of existing test suites*
- *Nevertheless, beyond the current limitations to the usability, the SEMuS tool seems promising for its effectiveness: all the generated test cases were correctly designed, in a way that they were able to detect software errors in the parts of the code whereas it was meant too*

### NEGATIVE COMMENTS

- *However, there are some typos in commands in SUM which can be corrected for error free configuration.*
- **Action taken:** SnT has improved the SUM.
- *If you run it on Windows machine with WSL or vagrant, there are chances you might run in few troubles with respect to versioning of different libraries/container/virtual env etc.*
- **Action taken:** SnT had not been able to replicate the problems; further investigation will be taken care of during the maintenance period. However, we recommend to use SEMuS only on UNIX systems.
- *Also creation of JSON file and test template can be it more difficult to scale especially when it is the case of microservices when there are large number of interacting microservices.*

- *But keep in mind that there is manual intervention of generating large amount of JSON and test templates which can be a time-consuming process in case of large software libraries.*
- **Action:** As detailed in Section 7, the two comments above can be targeted only by a dedicated follow-on activity.
- *However the tool is still in a very prototypical form.*
- *Due to the complexity of dependencies of E-SAIL software, the tool was able to working on a limited number of functions (the tool cannot compile files than depends on other files).*
- *Due to the previous limitation, the tool was able to generate the test cases only from the killed mutants, but not from the live mutants.*
- *The tool is not working with floating point variables. All these limitations make the SEMuS tool, in this preliminary version, inadequate to be deployed in any real development environment.*
- **Action:** As detailed in Section 7, the four comments above can be targeted only by a dedicated follow-on activity.

## 6 INTEGRATION WITH ECSS PRACTICES

To discuss applicability of mutation analysis and testing (either code-driven or data-driven) to different test levels (i.e., unit, integration, system, acceptance), we provide a generic overview of the interactions typically stressed by different test levels. Unit test cases focus on interactions within single units (e.g., functions belonging to a same source files) or few units belonging to a same component. Integration test cases trigger interactions between distinct units or multiple components. System test cases exercise interactions between all the components of the system. Figure 7 provides a generic overview of the interactions typically stressed by different test levels

According to ECSS standards, system test cases might be executed on a host system with emulated hardware. Acceptance test cases exercise all the system components but in the operational environment. For this reason, mutation analysis/testing cannot be adopted in the context of acceptance testing because the deployment of multiple version of the system (one for each mutant) might be particularly expensive and may lead to safety hazards.

*Unit testing* is the typical scenario in which code-driven mutation analysis and testing is adopted in other contexts, for this reason it should be targeted also in the case of space software. In addition, code-driven test generation approaches based on static program analysis can be adopted to automatically generate unit test cases. Data-driven mutation techniques, instead, are unlikely to be useful in the context of unit testing. Indeed, Unit test cases do not verify the interaction between units but rely on stubs when the testing of a unit requires the interaction with another component.

FAQAS has also shown that code-driven mutation analysis is feasible also in the context of *integration testing* and *system testing*. Code-driven test generation, instead, is not feasible in such contexts.

Integration and system test suite should also be the target of data-driven mutation analysis, which enables determining if relevant scenarios had been exercised (i.e., scenarios where all the possible message exchanges had been covered).

Depending on the development process, system-level test cases may focus only on specific features of the system under test; while unit test cases might be used only to cover exceptional cases. For this reason, each of these test suite may not reach 100% statement coverage. For the same reason, they may kill distinct subsets the mutants generated for the system. We suggest to compute the mutation score by considering all the available test suites (i.e., a mutant is killed if at least one test case of any available test suite fails).

Figure 8 shows the relationships between ECSS software testing practices and the main activities of the mutation analysis/testing process: code mutation, data mutation, code-driven test generation

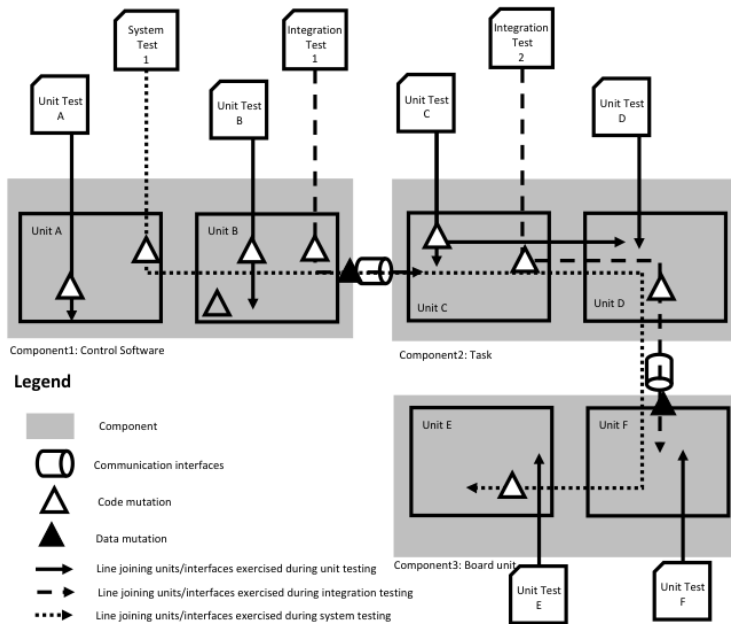


Fig. 7. Mutation Testing Approaches for Different Testing Levels.

(SEMuS), data-driven test generation (DAMTE), and inspection of mutation analysis metrics (to either evaluate test suite or derive test cases manually. These relationships guide the definition of a mutation testing process integrated with ECSS standards.

In Figure 8, black arrows show the specifications documents (i.e., Technical Specifications and Requirement Baselines) used to support ECSS testing activities (i.e., Unit Testing, Integration Testing, Validation activities with respect to the technical specification and Validation activities with respect to the requirements baselines). Colored arrows are used to associate ECSS activities to specific testing methods suggested in the ECSS standard (e.g., mission data is used for ECSS-E-ST-40C-5.6.4). Triangles are used to indicate which type of mutation testing (i.e., code-driven or data-driven) is likely applicable when a specific testing method is applied. White triangles are used to indicate that code mutation can be applied with a certain testing method, black triangles are used for data mutation.

Concerning the type of mutation activity associated to each testing method, we observe that code-driven mutation might be used for all the testing methods in use; sampling strategies will help code-driven mutation scale in the presence of long executions. Data-driven mutation, instead, is unlikely to be used with coverage based testing which often targets unit tests.

Test case generation based on symbolic execution can be used in the context of unit testing targeted by code-driven mutation (with SEMuS) and in the context of integration testing targeted by data-driven mutation (with DAMTE applied in the presence of specific architectures). For code-driven mutation, integration test suites can be improved only manually (because of the limitations of KLEE). System test suites can be improved only manually.

In Figure 8, dashed arrows show how the mutation testing procedures can contribute to ECSS activities. Overall, mutation analysis can be used to verify UT/IT/TS-RB test suites and guide engineers towards improving them (e.g., by selecting test inputs to kill mutants). Mutation testing may support the generation of unit test cases. Such support might be provided in both software (SW)

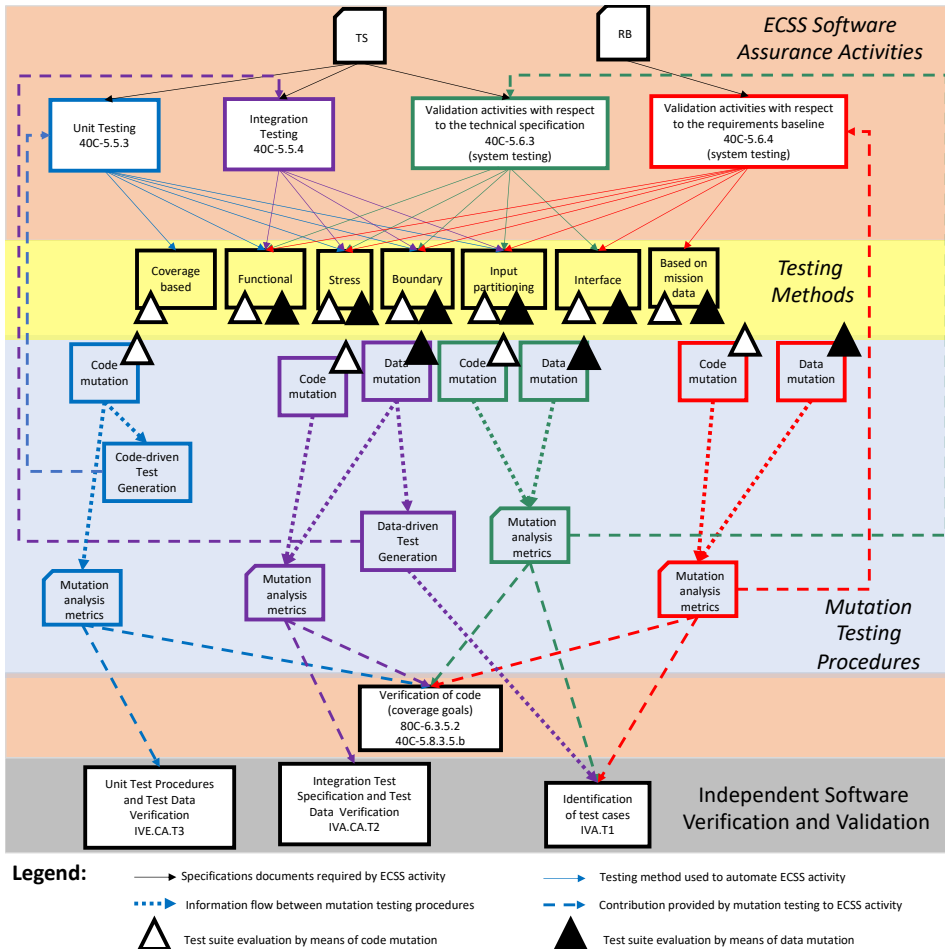


Fig. 8. Relations between ECSS activities and activities of the mutation testing process.

and ISVV life cycles. The mutation analysis metrics (e.g., mutation score) might be used to support SW verification activities; more precisely, they might be used as an additional coverage metric for the activities described in ECSS-Q-ST-80C 6.3.5.2 and ECSS-E-ST-40C 5.8.3.5.b. Also, mutation testing (i.e., automated test generation) supports the improvement of test sites. Independent Software Verification and Validation [13] can benefit from the mutation analysis/testing process as well. The mutation analysis metrics can support Unit Test Procedures and Test Data Verification (IVE.CA.T3 in [13]) and Integration Test Specification and Test Data Verification (IVA.CA.T2 in [13]). Finally, test generation and mutation score may support ISVV during the identification of test cases (IVA.T1 in [13]).

## 7 TOOLSET LIMITATIONS AND OPEN PROBLEMS

FAQAS led to developing tools that enable the application of mutation analysis and testing to space software. FAQAS mainly focused on selecting and extending existing preliminary research techniques to be applied in the space context. Also, FAQAS led to guidelines for the adoption of mutation analysis and mutation testing strategies within ECSS activities. However, FAQAS did not



address some mutation testing and analysis problems that are out of the scope of the project or can be considered open research problems:

- (1) FAQAS does not include methods to support an efficient integration of mutation analysis and testing techniques within software development and validation processes, which requires further research effort.
- (2) FAQAS cannot generate high-level mutants representative of errors in the understanding of software specifications.
- (3) The FAQAS test generation approaches require a great deal of manual effort for their configuration.

### 7.1 FAQAS Limitation 1 – Limited integration with software development practices

The integration of FAQAS mutation analysis techniques within space software development practices is currently limited by two factors: the lack of guidelines to identify a satisfactory mutation score and the lack of a strategy for the inspection of mutants. Procedures to determine an acceptable mutation score are necessary because it is unlikely to achieve a 100% mutation score. Indeed, the generation of new test cases to increase the mutation score has an associated cost, which prevents achieving a 100% mutation score. Since it is unlikely to achieve a 100% mutation score, defining guidelines that enable engineers to determine when it is acceptable to stop improving a test suite is necessary. Also, to adopt mutation analysis within ISVV practices, ISVV officers need similar guidelines to decide when a software test suite shall be considered adequate. Such guidelines may consist of threshold values to ensure different quality objectives.

Also, it shall be desirable to prioritize and reduce the live mutants to be inspected. For example, it shall be reasonable to maximize the diversity among the mutants to be inspected first thus increasing the likelihood of identifying different test suite shortcomings by inspecting a limited set of mutants. Also, since mutants may be redundant with each other, implementing a set of test cases that kill a diverse set of mutants shall likely enable the discovery of other mutants not selected for inspection – thus further increasing the mutation score. Identifying a diverse set of problems with a limited number of mutants to be inspected, would also enable ISVV officers to have a broad understanding of the problems affecting the test suite and help them evaluating the analyzed software artifacts.

### 7.2 FAQAS Limitation 2 – Lack of high-level mutants

The validation of FAQAS results performed by industry partners has shown that, frequently, the errors introduced by mutation operators are fine-grained corner cases, for which the implementation of a dedicated test case may appear unnecessary to software engineers. Indeed, under the assumption that software engineers are reliable and avoid simplistic mistakes, some of the faults injected by code-driven mutation operators (e.g., forgetting a clause in one expression) appear pessimistic and of little relevance to assessing the quality of test suites. Data-driven mutation analysis partially addresses the problem by introducing errors at a higher level (i.e., by altering the data exchanged by components); however, data-driven mutation analysis suffers from the opposite problem, the faults it simulates are too coarse (e.g., values out of range) and test suites likely discover them. Further, the literature lacks mutation operators that simulate specification-related errors (e.g., incorrect understanding of a sentence in the requirement specifications). We believe it is thus necessary to develop methods to drive mutation analysis based on software specifications. Specification-driven mutation analysis can be achieved in multiple ways. For example, for code-driven mutation analysis, it might be necessary to identify mutants that may reflect the misunderstanding of a requirement or the introduction of an articulate fault concerning multiple statements. For data-driven mutation

analysis, to increase its effectiveness, it is necessary to augment fault models with additional information about the system, for example, state-related data constraints. To this end, data-driven mutation analysis shall be combined with model-based testing artifacts (e.g., statecharts or timed automata).

### **7.3 FAQAS Limitation 3 – Limited test generation effectiveness with floating point instructions and external components**

The FAQAS code-driven mutation testing solution (i.e., the SEMuS tool) is limited by some peculiarities of KLEE, the state-of-the-art test generation tool integrated into SEMuS. Indeed, KLEE lacks support for floating-point computation and external, loosely coupled components. Also, FAQAS lacks a solution for data-driven mutation testing. Unfortunately, these limitations are open research problems for which industry-ready solutions do not exist. We briefly describe these problems in the following paragraphs.

First, KLEE does not properly support floating-point arithmetic; consequently, it cannot generate test cases killing the mutants for most mathematical functions used by space software. Prototype extensions of KLEE working with floating-point arithmetic support exist; however, their applicability to space software – and cyber-physical software more in general – remain to be assessed. Second, KLEE’s limitations concerning the processing of external, loosely coupled components derive from the symbolic execution process it relies on. Indeed, to derive the test inputs that kill a mutant, KLEE generates a symbolic formula that includes all the assignments observed in an execution path taken by the software under test. The formula is passed to a constraint solver, which looks for variables’ assignments that satisfy all the constraints in the formula (e.g., path conditions and assertions). In the presence of external libraries (e.g., calls to network functions or components without source code), KLEE cannot derive a formula that captures how the software computes its results; consequently, it may not be able to identify the desired inputs. Since data-driven mutants target the communication between external components, symbolic execution is inadequate for data-driven mutation testing too. Concolic execution (i.e., relying on concrete inputs collected during testing and symbolic inputs to be identified through constraint solving) is a solution that may help addressing the limitations of symbolic execution; however, for mutation testing [3], it has been evaluated only when testing opensource, bash programs, while its feasibility with large space software remains to be explored. Other valid alternatives are model-based test generation approaches, which are not affected by the limitations of static program analysis and symbolic execution [10].

### **7.4 FAQAS Limitation 4 – Limited test generation efficiency**

Another limitation of the FAQAS mutation testing solution is the need for the manual inspection of the generated test cases. More precisely, SEMuS, the FAQAS tool for mutation testing, requires engineers to verify that structured inputs and oracles are correctly set. This need for manual inspection depends on the nature of the targeted software (C and C++), which prevents the automated identification of input and output variables (e.g., a pointer variable might be used to provide input and outputs). Future work shall explore to possibility of relying on the available manually written test suites to automatically extract (e.g., through static program analysis) all the information required to generate executable test cases.

## **7.5 Future developments**

The above-mentioned limitations, might be addressed by a follow-up activity having the following objectives:

- *Objective 1: Support the integration of mutation analysis into space software development practices.* The activity shall develop a toolset that (1) determines thresholds for the mutation score that provide software quality guarantees (e.g., absence of severe failures, reduction of field failures, increased fault detection) and (2) select representative mutants to be inspected by engineers. The target users for the toolset shall be both engineers for space software companies and officers performing ISVV.
- *Objective 2: Specification-driven mutation analysis.* The activity shall extend code-driven mutation operators to generate higher-order errors, possibly simulating mistakes in the understanding of software specifications. Also, it shall extend data-driven mutation analysis with model-based support (e.g., to capture valid data ranges for specific program states).
- *Objective 3: Improve automated test generation effectiveness and efficiency.* The activity shall overcome the limitations of the test case generation tools used in FAQAS. To this end, the activity shall evaluate solutions available in the literature (e.g., to address floating point limitations) and leverage characteristics typical for the mutation testing context like the availability of seeds (e.g., the test cases that cover the mutated code without killing the mutant).

## 8 CONCLUSION

The FAQAS activity had been motivated by the need for high-quality software in space systems; indeed, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Before FAQAS there was no work on identifying and assessing feasible and effective mutation analysis and testing approaches for space software.

The main output of the FAQAS activity had been a toolset that implements three main features: code-driven mutation analysis (MASS), data-driven mutation analysis (DAMAt), code-driven mutation testing (SEMUS).

The empirical evaluation conducted with the aid of industrial case study providers have highlighted the practical usefulness of the FAQAS toolset, which led to the identification of relevant test suite shortcomings (test input partitions not exercised and missing test oracles) and bugs in the case study subjects. Since all the case study subjects considered in our empirical evaluation are space software systems that either undergo an extensive testing procedure and are deployed on orbit, the identification of such shortcomings and bugs indicate that the current procedures in place are not sufficient to guarantee software quality. Our results thus highlight the necessity for the adoption of an automated quality assessment toolset into development practices for space software — the FAQAS toolset has demonstrated to be an effective solution.

## REFERENCES

- [1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8. 209–224.
- [2] Thierry Titchou Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2020. SEMU: symbolic-execution based mutation testing. <https://github.com/thierry-tct/KLEE-SEMu>
- [3] Thierry Titchou Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. Killing stubborn mutants with symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–23.
- [4] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 597–608.
- [5] C. J. Clopper and E. S. Pearson. 1934. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika* 26, 4 (1934), 404–413. <http://www.jstor.org/stable/2331986>
- [6] GNU compiler collection. 2020. gcov?a Test Coverage Program. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>.
- [7] Oscar Eduardo Cornejo Olivares, Fabrizio Pastore, and Lionel Briand. 2021. Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain. *IEEE Transactions on Software Engineering* (2021), 1–1.

- <https://doi.org/10.1109/TSE.2021.3107680>
- [8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. <https://doi.org/10.1109/C-M.1978.218136>
  - [9] Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. 2015. Generating complex and faulty test data through model-based mutation analysis. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.
  - [10] Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. 2017. Augmenting field data for testing systems subject to incremental requirements changes. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 1 (2017), 1–40.
  - [11] European Cooperation for Space Standardization. 2009. ECSS-E-ST-40C ? Software general requirements. <http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/>
  - [12] European Cooperation for Space Standardization. 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/>
  - [13] European Space Agency. [n.d.]. ESA ISVV Guide issue 2.0, 29/12/2008.
  - [14] European Space Agency. 2017. ExoMars 2016 - Schiaparelli Anomaly Inquiry. *DG-I/2017/546/TTN* (2017). <http://exploration.esa.int/mars/59176-exomars-2016-schiaparelli-anomaly-inquiry/>
  - [15] Free Software Foundation. 2020. GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>
  - [16] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227.
  - [17] Farah Hariri and August Shi. 2018. SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation.. In *ASE*. 860–863.
  - [18] Marinis Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (aug 2018), 2426–2463. <https://doi.org/10.1007/s10664-017-9582-5>
  - [19] Daniel Kroening. 2021. Bounded Model Checking for Software. <https://www.cprover.org/cbmc/>
  - [20] LLVM. 2021. The LLVM Compiler Infrastructure. <https://llvm.org/>
  - [21] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.
  - [22] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. 2019. Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior. *IEEE Transactions on Software Engineering* 45, 9 (Sep. 2019), 919–944. <https://doi.org/10.1109/TSE.2018.2811489>
  - [23] Mozilla Foundation. [n.d.]. Peach Fuzzer, Opensource version. <https://github.com/MozillaSecurity/peach>
  - [24] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 354–365.
  - [25] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 537–548.
  - [26] Peach Tech. [n.d.]. Peach Fuzzer. <https://www.peach.tech>
  - [27] D. Shin, S. Yoo, and D. Bae. 2018. A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 914–931. <https://doi.org/10.1109/TSE.2017.2732347>
  - [28] Thanassis Tsiodras. 2020. Cover me! <https://www.thanassis.space/coverage.html>
  - [29] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 92–102.