securityandtrust.lu

Improve Mutation Testing in Space Software Systems (FAQAS-2)
Executive Summary Report (ESR)

ESTEC Contract Number 4000128969/19/NL/AS
ESA Technical Officer(s): Pedro Barrios, ESTEC, Noordwijk

Authors: Fabrizio Pastore
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg

FAQAS-2-ESR
Issue 1, Rev. 1
September 25, 2024

UNIVERSITÉ DU LUXEMBOURG

REVISIONS

Issue Number	Date	Authors	Description
FAQAS-2-ESR Issue 1 Rev. 0	September 3rd, 2024	Fabrizio Pastore	Initial release.
FAQAS-2-ESR Issue 1 Rev. 0	September 25th, 2024	Fabrizio Pastore	Addressed ESA comments; added Sections 6 and 7.

Contents

1 INTRODUCTION

The ESA activity *Improve Mutation Testing in Space Software Systems*, concerns the consolidation and extension of the work performed within the project *Applicability of Mutation Testing Method for Flight Software*, which was funded through ESA contract No. *4000128969/19/NL/AS*. The current activity led to the development of a framework and methodology referred to as *Fault-based Automated Quality Assurance* (FAQAS); for simplicity, we therefore refer to the two activities as FAQAS-1 and FAQAS-2.

Like FAQAS-1, FAQAS-2 is motivated by the **need for high-quality software in space systems**; actually, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Existing standards for space software development regulate software quality assurance and emphasize its importance. Mission failures due to insufficient software sanity checks are unfortunate examples of the necessity for systematic and predictable quality assurance procedures in space software. In general, software testing plays a prominent role among quality assurance activities for space software, and standards put a strong emphasis on the quality of test suites. For example, the European Cooperation for Space Standardization (ECSS) provides detailed guidelines for defining and assessing test suites. [9, 10].

Since one of the primary objectives of software testing is to identify 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. The approach is known as ***mutation analysis*** [19]. In mutation analysis, faults are injected automatically into the program through automated procedures called mutation operators. Mutation operators enable the generation of faulty software versions referred to as mutants. **Mutation analysis helps assess 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. A mutant leading to a test failure (i.e., at least one test case of the test suite fails when executed with the mutant) is said to be killed (i.e., detected) by the test suite. The output of mutation analysis is the input for ***mutation testing*** [19], which refers to the process of augmenting an existing test suite by deriving test cases that kill mutants not already killed by the test suite (i.e., live mutants). Mutation testing shall be supported by tools capable of identifying the inputs used in testing (e.g., through source code analysis). Mutation analysis concerns test suite assessment and mutation testing concerns test suite augmentation.

FAQAS-1 aimed to investigate mutation analysis as a mean to evaluate the quality of software test suites and mutation testing as a method to derive new software test cases in the context of space software (i.e., embedded software executed onboard, in-flight space systems). Before FAQAS-1, *there was no work on identifying and assessing feasible and effective mutation analysis and testing approaches for space software.* Space software is different from other types of software (e.g., Java graphical libraries or Unix utility programs); its characteristics prevent the adoption of well-known solutions from enhancing mutation analysis scalability, identifying mutants that are semantically equivalent to the original software, or redundant, and automatically generate test cases. First, space software typically contains many functions to deal with signals and data transformation, which may diminish the effectiveness of both compiler-based and coverage-based approaches to identify equivalent and redundant mutants. Second, space software is thoroughly tested with large test suites, thus exacerbating scalability problems. Third, it requires dedicated hardware, software emulators, or simulators, which affect the applicability of scalability optimizations that use multi-threading or other OS functions. The reliance on dedicated hardware, emulators, and simulators also prevent the use of static program analysis to detect equivalent mutants and automatically generate test cases.

FAQAS-1 led to the delivery of a ***toolset*** that includes the following tools:

- **MASS** (Mutation Analysis of Space Software, TRL 5), a configurable tool with a user manual that can be applied to whole software systems to perform ***code-driven mutation analysis***. Code-driven mutation analysis is a mutation analysis process that consists of creating faulty versions of the software under analysis by modifying its source code. MASS includes means to assure the scalability of the mutation analysis process by (1) restricting test case execution to those test cases that exercise the mutated lines of code, (2) sampling the mutants to be executed but relying on confidence estimation to compute an accurate mutation score, (3) prioritizing the order of execution of test cases. MASS was installed on third-party premises and independently used by third-party engineers in relevant cases.
- **DAMAT** (DAta-driven Mutation Analysis with Tables, TRL 4), a configurable tool with a user manual that can be applied to whole software systems to perform ***data-driven mutation analysis***. *Data-driven mutation analysis* alters the data exchanged by software components instead of mutating the implementation of the software under test. Data-driven mutation analysis enables the injection of faults that affect simulated components (e.g., sensors), which is not feasible with traditional, code-driven mutation analysis. It works by mutating, according to a fault model, the values assigned to selected data items in the messages exchanged by software components. DAMAT was installed on third-party premises and independently used by third-party engineers on the case study subjects of the project.
- **SEMuS** (Symbolic Execution-based MUtant analysis for Space software, TRL 3), a configurable tool, provided as an extension of MASS to perform test generation through symbolic execution [3]. Symbolic execution relies on the static analysis of the software under test, to determine, through constraint solving [2], the variable assignments that lead the software to certain states (e.g., reach a certain statement). SEMUs demonstrated effectiveness in generating test cases for standalone (e.g., not communicating through network) software units. SEMUs has been installed on third-party premises and independently used by third-party engineers on a subset of source files belonging to the case study subjects of the project.
- **DAMTE** (DAta-driven Mutation TEsting, TRL 2) is a prototype tool for the generation of test cases that detect data-driven mutants. DAMTE relies on symbolic execution to identify test inputs that trigger the generation of messages with data items containing values that enable the execution of data mutation. DAMTE enables the generation of inputs for data producers (in producer/consumer architectures) or client programs (in client/server architectures). DAMTE is provided as an extension of DAMAT; unfortunately, some manual effort and scaffolding is needed to apply it to new projects. DAMTE was applied to one case study subject of FAQAS-1.

FAQAS-2 aims at assessing the generalizability of MASS and DAMAT, the two most advanced tools in FAQAS-1, on new case study subjects, and improving their applicability, scalability, and effectiveness, based on the observed results. Further, FAQAS-2 aims at improving the effectiveness of FAQAS-1 test generation techniques (SEMuS and DAMTE), which are based on symbolic execution (an approach that, at a high level, requires the static analysis of the whole software under test) and hardly scale to complex software. Last, FAQAS-2 aims to provide guidelines and collect practitioners' feedback on applying mutation analysis and testing in the nominal ECSS process and ISVV context.

FAQAS-2 led to the delivery of the following ***toolset***:

- **MASS** (Mutation Analysis of Space Software, TRL 5), a consolidated version of the tool developed in FAQAS-1 that (1) improves usability through a simplified user interface, (2) increases reliability by repeating test executions in case of flaky test cases, (3) increase scalability by natively supporting parallel execution through the mutation testing of one file

a time, (4) extends applicability to systems that need to be executed in target environments that can only be monitored, where the full installation of MASS and its dependencies is not possible.

- **DAMAT** (DAta-driven Mutation Analysis with Tables, TRL 5), a consolidated version of the tool developed in FAQAS-2 that (1) enables handling of both little and big-endian architectures, (2) supports real-time platforms by delaying data recording to execution termination, and (3) reduces overall execution time by executing shortest test cases first.
- **MOTIF** (MUtation TestIng with Fuzzing, TRL 3), a new tool that performs mutation testing through fuzz testing [1, 17]. Fuzz testing consists of the the semi-random generation of inputs driven by a genetic algorithm guided by code coverage. MOTIF replaces SEMUs, and it demonstrated to be more effective than SEMUs since (1) it detects a higher number of mutants, and (2) it can be applied to a larger set of software systems because it does not inherit the limitations of symbolic execution.
- **DAMTEF** (DAta-driven Mutation TEsting with Fuzzing, TRL 2) a prototype tool for the generation of test cases that detects data-driven mutants and replaces DAMTE. DAMTEF relies on fuzzing instead of symbolic execution to identify test inputs that trigger the generation of the messages required for data mutation. DAMTEF outperforms DAMTE because it enables not only the generation of inputs for data producers (in producer/consumer architectures) or client programs (in client/server architectures) but, relying on fuzzing, can also generate inputs for mutations targeting messages produced by server programs or consumed by consumer or server programs. Like DAMTE, also DAMTEF is provided as an extension of DAMAT; it relies on a manual process to enable automated test input generation. DAMTEF has been applied to one case study subject of FAQAS-1.

Figure 1 provides an overview of FAQAS-2 tools and their dependencies.
In addition, ***FAQAS-2*** led to the following ***methodologies and guidelines***:

- a strategy to prioritise the inspection of mutants in order of relative code complexity (based on cyclomatic complexity) and speed up the identification of live mutants leading to the detection of faults;
- a method for the manual selection of code-driven mutants, thus enabling the application of code-driven mutation analysis;
- a method for the application of mutation analysis and mutation testing in contexts where software is developed in Java;
- guidelines for the application of the FAQAS-2 tools and determine test suite adequacy in ECSS nominal and ISVV contexts.

Last, FAQAS-2 achievements were ***disseminated*** in prestigious venues as follows:

- Presentation of the paper titled *Data-driven Mutation Analysis for Cyber-Physical Systems* [23], published on IEEE Transactions on Software Engineering, at the 45th IEEE/ACM International Conference on Software Engineering, May 14th 20th, 2023, Melbourne, Australia.
- Publication of the paper titled *DaMAT: A Data-driven Mutation Analysis Tool* [24], in the Proceeding of the 45th IEEE/ACM International Conference on Software Engineering, and presentation at the conference.
- Publication of the paper titled *Fuzzing for CPS Mutation Testing* [15], in the Proceeding of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23), and presentation at the conference.
- Publication of the paper titled *MOTIF: A tool for Mutation Testing with Fuzzing* [16], at the 17th International Conference on Software Testing Validation and Verification (ICST'24).

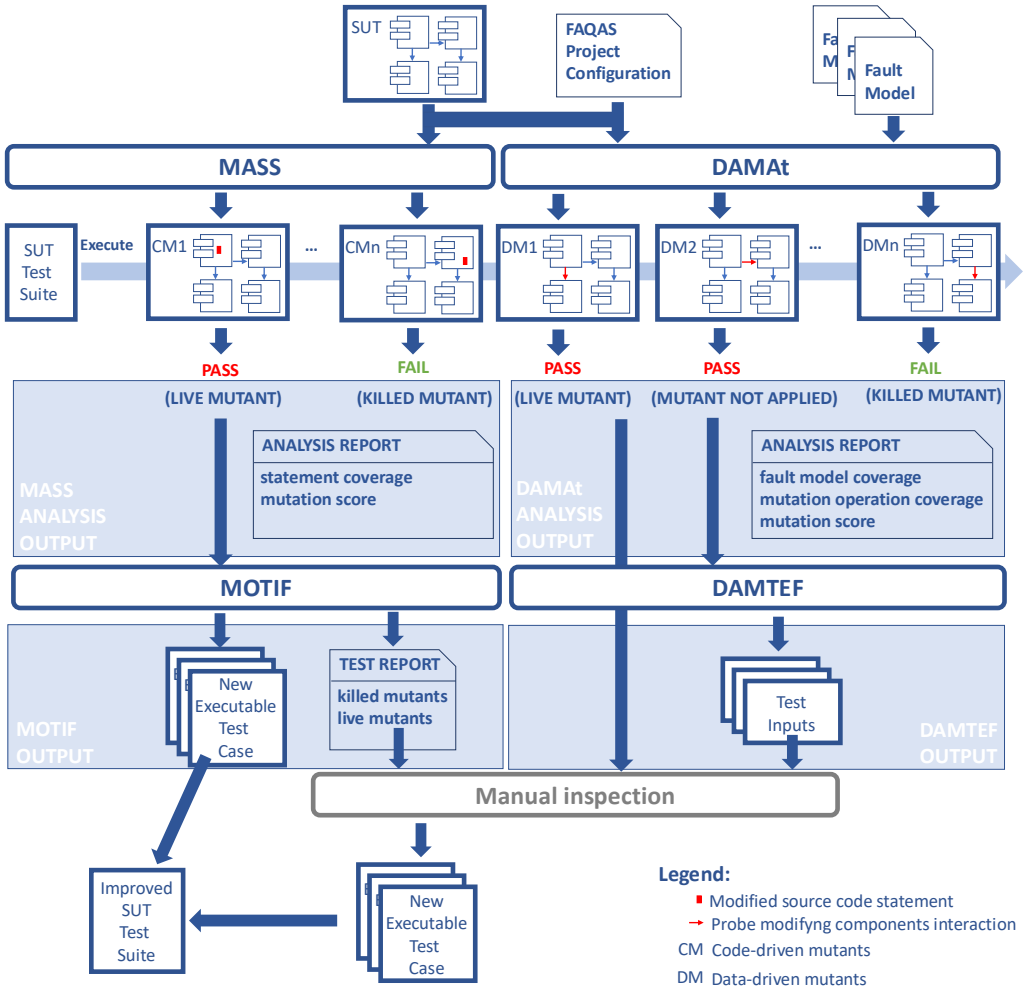

Fig. 1. Overview of the FAQAS-2 toolset

- Presentation of FAQAS-2 at ESA ADCSS workshop 2024.

1.1 Project achievements

This section summarises the activity requirements and the results achieved.

R1-1 The contractor shall identify the case study subjects to be used to validate the improved mutation analysis and testing process, and shall consider the following:

a) The selected case study subjects shall be provided together with data about the project development history. More precisely, the contractor shall provide development information extracted from versioning systems (e.g., Git repository backup) and issue tracking systems (e.g., a dump of Jira database).

b) The set of case study subjects shall extend the set of case study subjects already considered in FAQAS.

c) The selected case study subjects shall be representative of the diverse types of space software applications, including as a minimum embedded real-time software in the space segment. The contractor may also use other software used for empirical evaluation of software-testing research work.
d) The selected space software case study subjects shall have gone formal verification and validation according to ECSS software standards [AD01], [AD02] (developed as SW criticality A, B, or C), and coded in a typical language used in space applications (i.e., C and C++).
e) The Agency will also propose specific case studies (e.g. a SW module that is re- used across projects, such as the mathematical library LibmCS or a real-time operating system like RTEMS improvement/RTEMS-SMP; a full SW system, such the on-board software of an instrument control unit).
f) The selection of the case studies is to be agreed between the Agency and the contractor.

- *The project has been validated on a large set of case study subjects that are described in Section 3.*

R1-2 The contractor shall implement a methodology to efficiently identify test suite shortcomings based on mutation analysis. For example, the contractor should identify from the set of live mutants, a minimal subset to be inspected by engineers; such mutants shall represent diverse and critical shortcomings of the test suite. We define a critical shortcoming as a shortcoming that may prevent the identification of failures that may affect or partially affect the success of a mission.

- *SnT has empirically demonstrated that sorting mutants based on weighted cyclomatic complexity speeds up the identification of faults.*

R1-3 The contractor shall improve the FAQAS code-driven mutation testing tool (i.e., SEMuS); in particular, the following items shall be considered:
a) Since SEMuS requires the explicit specification of input and output variables; the considered solution shall automatically determine which variables are required to be inputs or outputs.
b) Evaluate the feasibility of solutions to enable test generation for programs relying on floating-point arithmetic.
c) Evaluate solutions (e.g., modelling of software components APIs) that enable applying symbolic execution when the software includes external libraries or loosely coupled components.

- *Instead of improving SEMUs, SnT has developed MOTIF, which outperforms MOTIF and overcomes its limitations.*

R2-1 The contractor shall implement a methodology for test adequacy determination based on code-driven mutation analysis; in particular, the following aspects shall be considered:
a) The methodology shall enable engineers to determine when the quality of a test suite is sufficient to ensure certain quality objectives, some examples include:
I. absence of severe failures in deployed software
II. reduction of field failures (i.e., failures in deployed software)
III. fault detection rate (FDR) higher than FDR achieved by test suites satisfying ECSS adequacy criteria (e.g., branch adequacy, MC/DC adequacy, code inspection)
b) Test suite assessment shall be based on the mutation analysis metrics (e.g., mutation score); the objective is to define thresholds above which the mutation score provides guarantees for specific quality objectives.
c) Additional mutation operators (e.g., higher order) might be considered.

- *SnT has empirically defined a methodology to determine test suite adequacy based on mutation analysis results, which is mentioned in Section 6.*

R2-2 The contractor shall define a methodology for ECSS standards and ISVV practices based on code-driven mutation analysis. The methodology shall include instructions to draft a verification report. The methodology shall be documented in the technical report [D3] "Integration of mutation testing within ECSS & ISVV practices".

- *The discussion on the adoption of FAQAS-2 methodology into ECSS Standards and ISVV are reported in Section 6.*

R2-3 The contractor shall improve the maturity of the tool for code-driven mutation analysis (i.e., MASS), considering the following aspects:

a) Since the current toolset does not provide interfaces for integrating the mutation analysis methodology into existing build pipelines (e.g., Jenkins, TASTE), the contractor shall define a test harness architecture that enables simple embedding of the MASS methodology as a part of a continuous integration and continuous development (CI/CD) pipeline.

b) Provide support to C++; MASS should support widely adopted C++ constructs (e.g., static variables).

- *The improvements made to MASS mainly concern: a) simplifying integration with Makefile-based process; however we do not cover build pipelines such as Jenkins or TASTE because not adopted by our partners. b) Provide support to C++; which has been demonstrated with the S5 case study.*

R2-4 The contractor shall prototype alternative solutions for code-driven mutation testing; such solutions shall aim at overcoming SEMuS limitations related to the generation of test cases for software using external libraries or loosely coupled components. Possible solutions include concolic execution and fuzzing [RD05].

a) The considered solutions shall be alternatives to what already implemented in FAQAS (i.e., differential symbolic execution).

b) Based on an empirical evaluation, the contractor should decide whether to integrate these alternative solutions into the tool developed by FAQAS (i.e., SEMuS) or to develop a newly independent tool.

R2-4 Section 2.2 reports on MOTIF:

(a) The execution of MOTIF with ESAIL demonstrates that the use of fuzzing is feasible also in the presence of external libraries.

(b) MOTIF deals with arrays and data structures

(c) MOTIF generates executable test cases including test inputs and assertions.

- *SnT has developed MOTIF, a tool relying on fuzz testing as an alternative to symbolic execution (i.e., the core of SEMUs) for the generation of test cases that kill mutants. MOTIF demonstrated to be more effective than SEMUs although some complementarities had been identified. MOTIF does not require the specification of input and output variables thus addressing* R1-3-a. *Also, MOTIF can generate inputs for floating point variables thus addressing* R1-3-b. *MOTIF can work with software that includes external libraries or loosely coupled components (i.e.,* R1-3-c*).*

R3-1 The contractor shall improve the data-driven mutation analysis methodology, addressing the following concerns:

- a) Simplify the applicability of the technique to case studies by reducing engineers' manual intervention. For example, by proposing a solution that do not require the definition of a fault model.
- b) The methodology shall support different types of data structures as required in the case study subjects defined by [R1-1].

- *SnT applied DAMAT to BepiColombo to demonstrate the generalizability of the approach. To enable the application of DAMAT to BepiColombo, SnT integrated the following improvements:*

  - *a) deferred logging of mutation information, to cope with real-time-requirements of SUT;*
  - *b) support for Little Endian and Big Endian architectures;*
  - *c) improved methodology to allow for a quicker mutation testing process when traceability information is provided with test cases.*
- *Also, SnT has defined a solution that enables applying DAMAT on Java systems. It relies on JNI to mutate Java data.*

R3-2 The contractor shall prototype solutions for data-driven mutation testing to cover the mutants not killed by the existing test suite, addressing the following concerns:
  - a) Implement an automated solution for generating test templates to drive the test input generation.
  - b) Integrate a solution to deal with loosely coupled components and external libraries.
- *SnT has developed DAMTEF. It has been applied on GSL's libParam, which was used for DAMTE, and demonstrated to outperform the latter.*

R4-1 The contractor shall implement a methodology for test adequacy determination based on data-driven mutation analysis. The development shall be consistent (e.g., provide guidelines for drafting verification reports) with requirement [R2-1].
- *Contributions are discussed in Section 6.*

R4-2 The contractor shall define a methodology for ECSS and ISVV practices based on data-driven mutation analysis. The development shall be consistent (i.e., cover the same aspects) with requirement [R2-2].
- *See Section 6.*

R4-3 The contractor shall improve the maturity of the data-driven mutation analysis tool. This requirement includes integrating the mutation analysis tool (i.e., DAMAT) and the mutation testing tool (i.e., DAMTE) into existing build infrastructures (e.g., TASTE, Jenkins).
- *SnT improved DAMAT as follows:*
  - *Introduced a feature to test a subset of mutants and test cases.*
  - *Introduced an optimization feature to store mutation operation / fault model coverage information on demand (based on a signal received by the SUT).*
  - *An error-handling system was introduced to facilitate the use of the tool and to help fix configuration problems.*

R4-4 The contractor shall improve the maturity of the mutation testing tool. This requirement includes integrating the mutation testing toolset (i.e., SEMuS and DAMTE) into existing build infrastructures (e.g., TASTE, Jenkins).
- *SnT conducted experiments to determine the best fuzzing configuration for MOTIF.*
- *SnT has integrated MOTIF and MASS, MOTIF configuration can now be generated after the execution of MASS.*

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

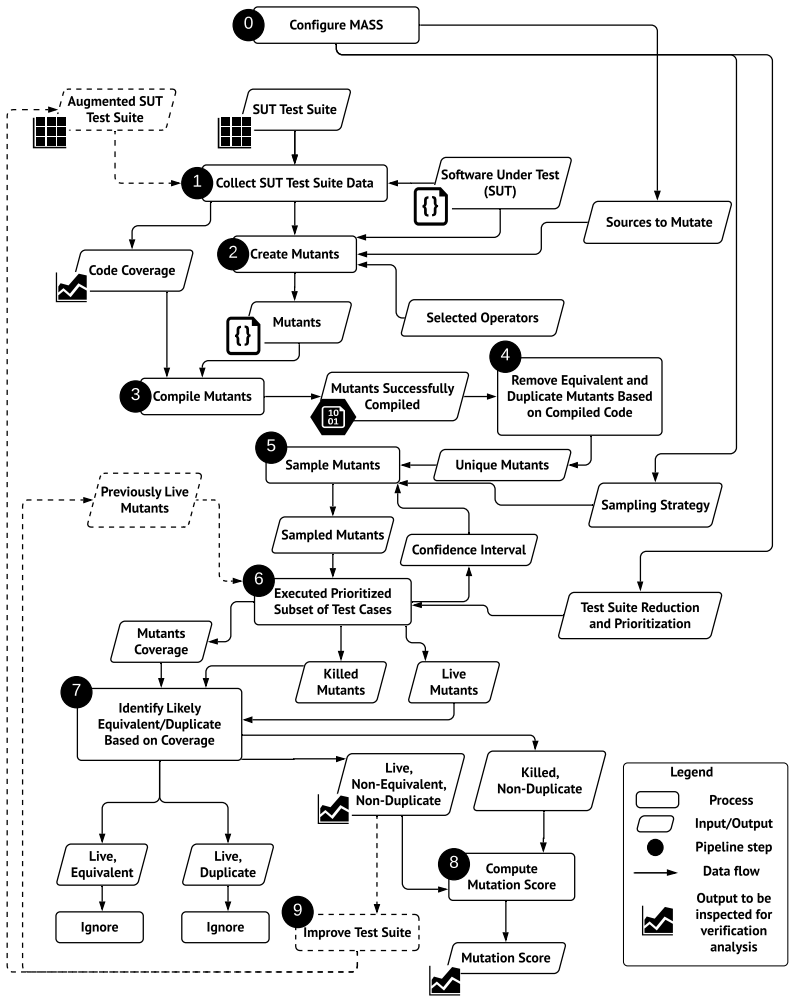

Fig. 2. Overview of the MASS workflow

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 [7] and GDB [12], enabling the collection of coverage information for embedded systems without a file system [22].

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.

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	$\{(op_1, op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\}\}$ $\{(e, !(e)) \mid e \in \{\texttt{if(e)}, \texttt{while(e)}\}\}$
	SDL	$\{(s, \texttt{remove}(s))\}$
	UOI	$\{(v, -v), (v, v-), (v, ++v), (v, v++)\}$
OODL	AOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{+, -, *, /, \%\}\}$
	LOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&\&, ||\}\}$
	ROD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{>, >=, <, <=, ==, !=\}\}$
	BOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&, |, \wedge\}\}$
	SOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\gg, \ll\}\}$
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 [14]. 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).

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.

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. [25], who compared eight strategies for sampling mutants. The ***uniform fixed-size sampling*** strategy stems from the work of Gopinath et al. [13] 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 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 [6].

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):

$$MS = \frac{|KND|}{|LNEND| + |KND|} \tag{1}$$

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 [20]; 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].

```
##### MASS Output #####
## Total mutants generated: 28071
## Total mutants filtered by TCE: 6918
## Sampling type: fsci
## Total mutants analyzed: 461
## Total killed mutants: 369
## Total live mutants: 92
## Total likely equivalent mutants: 53
## MASS mutation score (%): 90.44
## List A of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_a
## List B of useful undetected mutants: /opt/MLFS/RESULTS/useful_list_b
## Number of statements covered: 1973
## Statement coverage (%): 100
## Minimum lines covered per source file: 2
## 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 automated through SEMuS . It consists of deriving test inputs that kill live mutants.

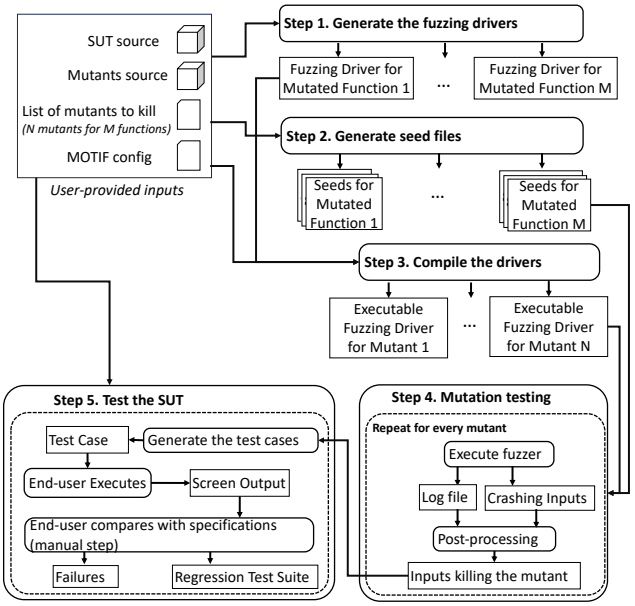

Fig. 3. The MOTIF process.

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 MOTIF is feasible, we suggest relying on MOTIF to automatically generate test cases for all the mutants appearing in *useful_list_a* and in *useful_list_b*.

2.2 MOTIF

MOTIF is a tool for code-driven mutation testing that relies on fuzzing; it's workflow is shown in Figure 3. MOTIF is started by running its Python entry point in a directory selected by the end-user as the MOTIF workspace. The MOTIF workspace structure is shown in Figure 4; it includes all the inputs required by MOTIF , which are the SUT source code, the mutants source code, a configuration file for MOTIF , and a text file with the names of the mutants to kill. We rely on the mutants generated by MASS.

MOTIF creates a directory where mutation testing outputs are stored (*outputs* in Figure 4). Its sub-directories contain the outputs generated by each step of the MOTIF workflow, distributed in one additional sub-directory for each mutant or mutated function.

In Step 1, MOTIF relies on the *clang* static analysis library [5] to build an abstract syntax tree of the SUT and determine the types of parameters required by each mutated function. Such information is used to generate a driver for mutation testing (fuzzing driver); an example is shown in Figure 5.

Fig. 4. Structure of the directory for a MOTIF project

```
WORKSPACE/ #dir chosen by the end-user, any name is good
├── repos/      #dir created by MOTIF where the SUT will be compiled
├── outputs/    #dir where mutation testing outputs are stored
│   ├── 1-func-drivers/ #dir with fuzzing drivers for mutation testing
│   │   ├── a_moudle.a_function/ #dir with the drivers for a mutated function
│   │   └── <..>/       #sub dirs, one per mutanted function
│   ├── 2-func-inputs/  #dir with seed inputs generated by MOTIF
│   │   └── <..>/       #sub dirs, one per mutanted function
│   ├── 3-mutant-funcs/ #dir with the mutants processed by MOTIF
│   │   └── <..>/       #sub directories, one per mutant
│   ├── 4-mutant-bins/  #dir with the compiled fuzzing drivers
│   │   └── <..>/       #sub dirs, one per mutant
│   ├── 5-fuzzing/      #dir with the outputs of the fuzzer
│   │   └── <..>/       #sub dirs, one per mutant
│   └── 6-testcase/     #directory with the generated test cases
│       └── <..>/       #sub dirs, one per mutant
├── mutants.tar  #tar with all the mutants
├── src.tar      #tar with the source code project of the SUT
├── live_mutants #text file including the list of mutants to kill
└── config.py    #config file for MOTIF
```

Fig. 5. Example fuzzing driver for the ASNLib subject.

```
1  int main(int argc, char** argv){
2    load_file(argv[1]);  // load the input file and extends it if needed
3    /* Variables for the original function */
4    T_POS origin_pVal;   // for the first parameter
5    int origin_pErrCode; // for the second parameter
6    /* Variables for the mutated function */
7    T_POS mut_pVal;      // for the first parameter
8    int mut_pErrCode;    // for the second parameter
9    /* Variables for the return values */
10   flag origin_return;  // for the original
11   flag mut_return;     // for the mutant
12   /* Copy the input data to the variables for the original function */
13   get_value(&origin_pVal, sizeof(origin_pVal), 0);
14   get_value(&origin_pErrCode,sizeof(origin_pErrCode),0);
15   log("Calling the original function");
16   origin_return = T_POS_IsConstraintValid(&origin_pVal, &origin_pErrCode);
17   /* Copy the same input data to the variables for the mutated function */
18   seek_data_index(0); //reset the input data pointer
19   get_value(&mut_pVal, sizeof(mut_pVal), 0);
20   get_value(&mut_pErrCode, sizeof(mut_pErrCode), 0);
21   log("Calling the mutated function");
22   mut_return = mut_T_POS_IsConstraintValid(&mut_pVal, &mut_pErrCode);
23
24   log("Comparing result values: ");
25   ret += compare_value(&origin_pVal, &mut_pVal, sizeof(origin_pVal));
26   ret += compare_value(&origin_pErrCode,&mut_pErrCode, sizeof(origin_pErrCode));
27   ret += compare_value(&origin_return, &mut_return, sizeof(origin_return));
28
29   if (ret != 0){
30     log("Mutant killed");
31     safe_abort();
32   }
33   log("Mutant alive");
34   return 0;
35 }
```

The fuzzing driver declares two sets of variables (Lines 4-5 and 7-8) that are provided as input to the original and to the mutated function, respectively. They are assigned with a byte-by-byte copy of the same portion of the input file provided by the fuzzer (Lines 13-14 and 19-20); the copied bytes match the size of the assigned variable.

Fig. 6. Example test case for the ASNLib subject.

```
1  /*Variables with the data in the file killing the mutant*/
2  const char input_data_pVal={0xFF,0xFF,...,0xFF};
3  const char input_data_pErrCode={0xFF,0xFF,0xFF,0xFF};
4  /*Variables with the values observed after mutation testing*/
5  const char expected_pVal={0xFF,0xFF,...,0xFF};
6  const char expected_pErrCode={0xFF,0xFF,0xFF,0xFF};
7  const char expected_return={0x00,0x00,0x00,0x00};
8  /*Test case*/
9  int main(int argc, char** argv){
10   T_POS pVal;
11   int pErrCode;
12   int _return;
13   /*Initialize inputs*/
14   memcopy(&pVal, input_data_pVal, sizeof(pVal));
15   memcopy(&pErrCode, input_data_pErrCode, sizeof(int));
16   /* Invoke the original function*/
17   _return = T_POS_IsConstraintValid(&pVal, &pErrCode);
18   /* Print output values of the original function */
19   printf_struct("pVal (T_POS)=", &pVal, sizeof(pVal));
20   printf("pErrCode (int) = %d\n", pErrCode);
21   printf("return (flag) = %d\n", _return);
22   /* Generated assertions enabling  regression testing*/
23   assert( 0==compare((char*)&pVal, sizeof(pVal)));
24   assert( 0==compare((char*)&pErrCode, sizeof(pErrCode)));
25   assert( 0==compare((char*)&_return, sizeof(_return)));
26   return 0;
27 }
```

The original and the mutated functions are then invoked (Lines 16 and 22). The fuzzing driver then compares the output generated by the original and the mutated functions (Lines 25-27). Since in C, thanks to pointer and reference arguments, every parameter may be used to store outputs, we compare all the parameters and return values of the original and mutated functions; our choice cannot lead to inaccurate identification of killed mutants because, by definition, input parameters are not modified. For pointers, we compare the pointed data (e.g., an `int` instance for `int*`).

When the outputs differ, the fuzzing driver stops its execution with an abort signal (Line 31) thus letting the fuzzer detect the aborted execution and store the input file.

In Step 2, MOTIF generates seed files based on the types of input parameters for the function under test. The generated files contain enough bytes to fill all the input parameters with values covering basic cases. Precisely, for each primitive type, we have identified three seed values that are representative of typical input partitions [15]. For example, for numeric values, we provide zero, a negative, and a positive number. For each fuzzing driver, MOTIF generates at most three seed files in such a way that every parameter of the function under test is assigned with each seed value at least once.

In Step 3, MOTIF compiles the fuzzing driver, the mutated function, and the SUT using the fuzzer compiler; compile commands can be specified in MOTIF 's configuration.

In Step 4, MOTIF runs the fuzzer. The execution leads to the generation of fuzzing driver logs and crashing inputs. MOTIF processes the corresponding logs to identify likely killed mutants, which happens when either the execution aborts because the generated outputs differ for the original and the mutated function, or there is a crash in the execution of the mutated function. Inputs leading to apparently killed mutants are further processed to determine if the function under test generates non-deterministic outputs: MOTIF executes them with an additional fuzzing driver, automatically generated, that executes the original function twice. If outputs differ, then the function under test is non-deterministic and the mutant has not been killed.

In Step 5, MOTIF generates a unit test case that is similar to the fuzzing driver created for the same function under test, an example is shown in Figure 6. Different from the fuzzing driver, the generated test case declares a set of arrays initialized with data taken from the file that killed the mutant (Lines 2-3, we use hexadecimal values); each array contains the bytes used by the fuzzing driver to initialize one variable. Also, it declares a set of arrays (Lines 5-7) initialized with the data observed, during the mutation testing post-processing step (Step 4), after the execution of the original function. If the original function is not faulty, the data observed after the execution of the original function can be used in test assertions for regression testing.

The `main` function initializes the variables passed to the function under test (Lines 14-15) and then, after invoking the function under test (Line 17), prints out the variables' values (Lines 19-21). Such print instructions are necessary because the end-user should verify if the output values are correct; otherwise, a fault has been found. Finding such faults is a key benefit of mutation testing, which enables the detection of actual faults by exercising the software with inputs generated for injected faults. If no fault has been detected, then the generated test case can be reused as-is for regression testing; indeed, the test case includes assertions (Lines 23-25) automatically verifying that the output variables match the expected ones.

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 [8, 18, 21]; 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 8. In Step 1, based on a methodology provided with the DAMAt documentation, the engineer prepares a fault model specification tailored

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 (Δ).
Value below threshold (VBT)	Replaces the current value with a value below the threshold T for a delta (Δ).
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 *VALUE*.
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 Δ from it.
Signal Shift (SS)	The mutated value is derived by adding a value Δ 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 Δ.
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.

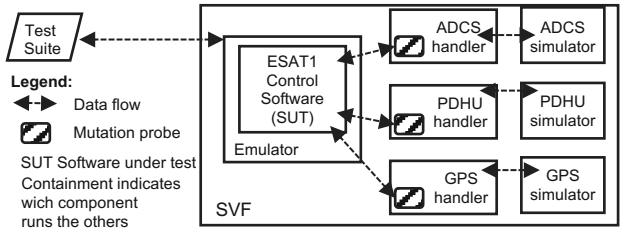

Fig. 7. Data mutation probes integrated into ESAIL.

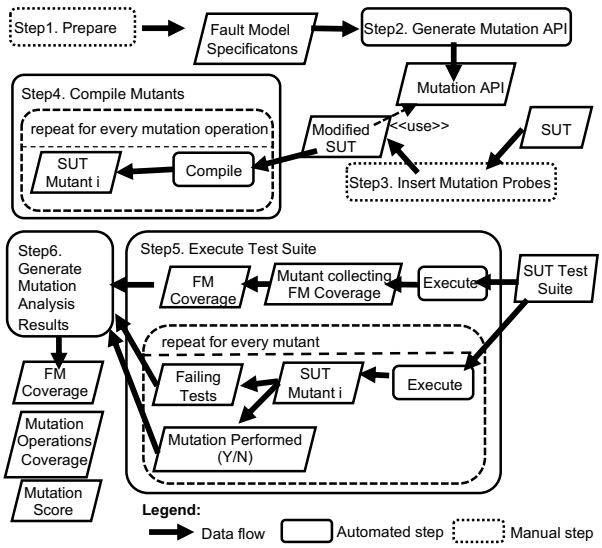

Fig. 8. The *DAMAT* process.

to the SUT. Our methodology enables the specification of all possible interoperability problems in the SUT while minimizing equivalent and redundant mutants.

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 where not discovered by the test suite.

2.4 Data-driven Mutation Testing: DAMTEF

DAMTEF 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 9). Different from DAMTE , it can generate inputs to support data-driven mutation analysis with porbes applied on the server-side or on the receiving side of the client.

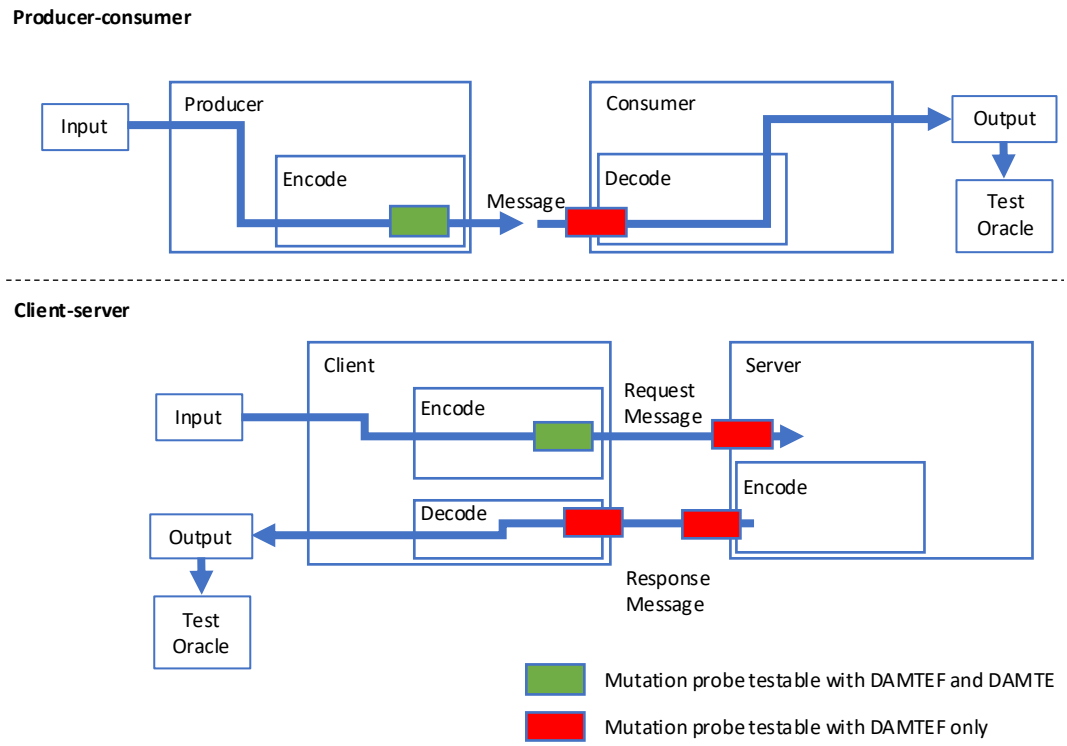

Fig. 9. DAMAT: mutation when encoding the request.

To generate the required inputs we rely on an ***extended data mutation probe***. The extended data mutation probe relies on *DAMAT* to track when the mutation probe has been successfully appplied; combined with a fuzzer for test input generation, it enables mutation testing.

3 CASE STUDIES

The FAQAS-2 toolset has been applied to six case study systems: ESAIL, *LIBGCSP*, *LIBParam*, *LIBUTIL*, MLFS, ASN1SCC, BepiColombo SIXS/MIXS onboard software, S5 L1bPP, ExoMars Recovery Software Image, Z80 emulator, and Zynq-7000 SoC SW.

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***[1] (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[2] 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.

The ***BepiColombo SIXS/MIXS onboard software*** (hereafter, BepiColombo, for brevity) runs in the combined Data Processing Unit (DPU) of the SIXS/MIXS instruments. The DPU is used to control instrument power and operating states, to monitor instrument operations and to handle telecommand and telemetry communications. Another responsibility of the SIXS/MIXS DPU is to convert scientific data from a very high number of individual detections to more manageable summary information, such as periodic summary counters, spectra, and histograms. Huld developed and verified the onboard software in accordance with ECSS standard (category C, tailored to the project). The flight code has been developed in C and corresponds to approximately 30k LOC.

The ***S5 UVNS L1bPP*** (S5 L1bPP) project has been developed by Huld, it consists of the L1b prototype data processing software (a data processor), as part of the Sentinel-5 mission. S5 L1bPP has been developed and verified in accordance with ECSS standard (category D). It is developed in C++ and contains around 23k LOC.

The ***ExoMars Recovery Software Image*** (hereafter, ExoMars, for brevity) is the back up software which provides basic functionality to the ground to investigate and maintain the ExoMars Rover Vehicle Software (RVSW). RSI implements basic communication facilities with ground such as housekeeping telemetry and memory management to allow investigation and maintenance to be performed on the RVSW images. RSI has been developed and verified in accordance with ECSS standard (category B). It is developed in C and contains around 11,5k LOC.

The ***Z80 emulator*** is an emulator of Zilog Z80, an 8-bit processor widely used in the 80s. The Z80 Emulator replicates the behaviour of part of the Z80 microprocessor on the host machine. This subject has been introduced by LuxSpace to conduct an independent validation of MASS, DAMTE, and MOTIF.

The ***Zynq 7000 SoC SW*** is a simple program to be executed in the Cortex-A9 processor inside Zynq 7000 SoC. The Zynq-7000 SoC (for SCSW) and SAMV71 microcontroller (for BOSS) are used to test software for the Triton-X project. This subject has been introduced by LuxSpace to conduct an independent validation of MASS with hardware-in-the-loop.

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. Column *Owner* provides the name of the case study subject provider. Column *Test suite* provides an indication of the test suite type (unit, integration, system). Column *Fault records* indicates if the case study was provided with a list of fault descriptions. Column *Assessed tools* indicates what FAQAS tool has been validated with the case study (we also include SEMUs because experiments had been re-executed for comparison

[1] https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software
[2] https://github.com/ttsiodras/asn1scc

Table 3. Overview of subject systems and tool assessment.

Owner	Subject	LOC	Test suite	Fault records	Assessed tools				
					MASS	SEMUs	MOTIF	DAMAT	DAMTEF
LuxSpace	ESAIL-CSW	74,155	System + Unit	Yes	Yes*	-	Yes	Yes*	-
	Z80 emulator†	1999	Unit	-	Yes	-	Yes	Yes	-
	Zynq 7000 SW†	20	Unit	-	Yes	-	-	-	-
GomSpace	LibGCSP	9,836	Integration	Yes	Yes*	Yes	Yes*	-	-
	LibParam	3,179	Integration	Yes	Yes*	-	-	Yes*	Yes
	LibUtil	10,576	Unit	Yes	Yes*	Yes	Yes	Yes*	-
	HOOP	27,000	System	Yes	-	-	-	-	-
ESA	MLFS	5,402	Unit	Yes	Yes*	Yes	Yes	-	-
	ASN1.CC	4,338	Unit	Yes[a]	Yes*	Yes	Yes	-	-
Huld	BepiColombo SIXS/MIXS	30,000	System + Unit	Yes	Yes	-	-[b]	Yes	-
	S5 L1bPP	23,000	Unit + Integration	Yes	Yes	-[c]	Yes	Yes	-
	ExoMars RSI	11,500	Unit + System	Yes	Yes	-	-	-	-

[a]SnT will rely on faults identified during FAQAS-1.
[b]Not feasible to apply MOTIF because BepiColombo relies on old environment not feasible to process with MOTIF.
[c]Not feasible because SEMUs does not support C++.
* Experiments performed in the context of FAQAS-1
† Subject internal to LuxSpace, used to support the independent assessment of the tools.

with MOTIF). Notably, MASS, MOTIF, and DAMAT had been assessed by LuxSpace completely independently, on subjects not delivered to SnT.

4 EMPIRICAL EVALUATION

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

4.1 MASS

MASS has been applied to the new case study subjects introduced in FAQAS-2, they are BepiColombo, S5 L1bPP, and ExoMars. In addition, MASS had been independently assessed by LuxSpace on two internally developed subjects, *Z80 emulator* and *Zynq 7000 SoC*. Table 4 provides an overview of the results; in general, it has been feasible to apply MASS to the selected case studies (some additional features had been introduced for the purpose), and all the live mutants have shown to provide useful information about the limitations of the test suites.

4.2 MOTIF

Our empirical evaluation demonstrated the scalability and effectiveness of MOTIF for space software. Our results in Table 5 show that MOTIF kills between 40% and 89% of the live mutants, contributing to increasing the mutation score by 7 to 37 percentage points. Our results also demonstrated the practical usefulness of MOTIF. Indeed, MOTIF enabled the identification of four faults in one case study. Also, the generated test cases concerned inputs that are relevant (according to specifications) but not tested by the test suites. Further, we performed an extensive empirical assessment of

Table 4. MASS results with FAQAS-2 subjects.

Subject	Mutants								
	Sampling type	Generated	Eliminated by compilation + TCE	Useful	Tested	Killed	Live	MS	Notes
Huld - Bepi-Colombo	Stratified	12318	11090	1228	206	196	10	95.61%	To handle this subject, MASS has been extended to deal with execution environments different than the one running MASS. One of the 206 mutants was live (205 for MS computation). All mutants led to the identification of useful test suite improvements.
Huld - S5 L1bPP	Uniform	42198	20072	22126	2236	1434	802	64.13%	MASS has been extended to handle coverage on C++. Out of 10 manually inspected mutants, all led to the identification of useful test suite improvements.
Huld - Exo-Mars RSI	Manual	14306	5702	8604	10	8	2	80.00%	MASS has been extended to enable the manual selection of mutants without code coverage information. Out of 10 manually inspected mutants, all led to the identification of useful test suite improvements.
LuxSpace - Z80 emulator	FSCI	7582	3297	4285	502	389	113	77.49%	No extension of MASS was needed. Out of 10 manually inspected mutants, all led to the identification of useful test suite improvements.
LuxSpace - Zynq 7000 SoC	ALL	39	2	37	37	15	22	40.54%	The case study demonstrated feasible to collect coverage from target hardware and provide it to MASS. Out of 3 manually inspected mutants, one was equivalent to the original program, the others led to the identification of useful test suite improvements.

Legend for table columns: *Sampling type*, type of sampling applied by MASS. *Generated:* number of mutants created by MASS. *Eliminated by compilation + TCE:* number of mutants eliminated either becaus ethey do not compiler or are equivalent/duplicate according to Trivial Compiler Equivalence detection. *Useful:* number of remaining mutants. *Tested:* number of mutants tested by MASS with the test suite of the SUT. *Killed:* number of mutants killed by the test suite. *Live:* live mutants. *MS:* mutation score. *Notes:* additional notes.

MOTIF to determine its best configuration, which consists of relying on Clang, the LAF compiler optimization, and the ASAN address sanitizer.

Table 5. MOTIF results.

Subject	Open-source	LOC	# Test cases	Statement coverage	Test suite mutation score	Live mutants	% Mutants killed by MOTIF	Improved mutation score	
LIBUTIL	No	10,576	201	83.20%	71.20%	443	49.84%	85.55%	(+14.35)
ASNLib	Yes	7,260	139	95.80%	58.31%	1,347	88.74%	95.31%	(+36.99)
MLFS	Yes	5,402	4,042	100.00%	81.80%	3,891	39.85%	89.05%	(+7.26)
ESAIL	No	2,235	384	95.36%	65.36%	581	39.5%	79.04%	(+13.68)
S5	No	54,696	36	62.23%	64.13%	99	82.53%	93.72%	(+29.60)

4.3 DAMAt

Table 6 shows the mutation analysis results of DAMAT. In general, our results confirm the generalizability of the approach, which can be applie dto very different case study subjects from BepiColombo to Z80. In most of the cases, it was possible for partners to confirm the correctness of the results, although the lack of coverage was often due to the selection of a test suite subset for the experiments.

Table 6. Data-driven mutation analysis results.

Subject	# FMs	FMC	#MOs-CFM	#CMOs	MOC	Killed	Live	MS
BepiColombo - Mutation of Periodic Telemetries	10	100.00%	93	89	95.70%	89	89	100.00%
BepiColombo - Mutation of Telecommand Headers	8	50.00%	41	40	97.60%	40	36	90.00%
BepiColombo - Mutation of Telecommand 198	13	100.00%	59	55	93.20%	55	29	52.70%
LXS Z80 emulator	1	100.00%	19	19	100%	19	19	100%

FM=Fault Model, FMC=Fault Model Coverage, MOs-CFM=Mutation Operations in covered FMs, CMO=Covered Mutation Operation, MOC=Mutation Operation Coverage, Killed=Number of mutants killed by the test suite, Live=Number of mutants not killed by the test suite, MS=Mutation Score. The mutation score for *LIBGCSP* is not available because of nondeterminism observed while running the experiments.

4.4 DAMTEF

DAMTEF aims to address a task (i.e., test generation at the 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, we assessed only the feasibility of DAMTE.

We relied on DAMTEF to generate inputs for the *LIBParam* client API functions. Such inputs enable the exchange of messages between the *LIBParam* client and the *LIBParam* server. Our results demonstrated that DAMTEF can successfully generate test inputs to enable the application of two out of three live mutants, thus outperforming DAMTE, which generated results only for one of them. The mutant not enabled by DAMTEF is intractable because would require a faulty implementation in order to be able to perform data mutation. For the two mutants enabled by DAMTEF, we performed an extensive assessment of the approach with 200 executions, and report that it successfully generate results in 89% of the cases.

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 the generation of fuzz drivers, which are key for the generation of test inputs. In the future, large language models might address this problem.

5 INDUSTRIAL VALIDATION SUMMARY

Below we report verbatim the positive and negative comments provided in the validation deliverables of the project by our industry partners. In general, positive comments concern the effectiveness of MASS and DAMAT in identifying test suite limitations, and the capability of MOTIF to generate useful test cases. Negative comments, concern mainly the configuration of the tools, which require some effort because they need to inherit all the information about the software under test.

5.1 Overall Comments

POSITIVE COMMENTS

ALL *The analysis of the results of the toolsets (MASS, MOTIF) shows a clear effectiveness in detecting/generating cases for the test suites in different environments: different programming languages, different testing frameworks.*

MASS *For evaluation purposes, the test suite of the Z80 emulator has been modified to remove some relevant testcases. MASS tool has been able to detect all these cases, as seen in the results. Furthermore, the useful list generated by the tool is showing the most representative cases of these intentional errors.*

MASS *the application of MASS is perfectly possible to be used in hardware on the loop with a dummy program. This opens the doors to the verification of more complex systems.*

MASS *limitations with the executed test suites were correctly identified by MASS.*

MASS *A promising approach based on selecting mutants based on high cyclomatic numbers of the mutated methods was applied by SnT for ExoMars to reduce the time required by test execution.*

DAMAT *the use of DAMAT can be complementary to MASS and other tools.*

DAMAT *The configuration and use of the tool (DAMAT) has been easier and more straighforward than the performed in MASS/MOTIF. The documentation, like that of these toolsets, is excellent, with good and detailed use cases. Furthermore, the support from the SnT team has been very good and efficient.*

DAMAT *The installation of DAMAT was relatively straightforward into old test environment.*

MOTIF *Regarding the evaluation of the MOTIF tool, the results are also very satisfactory. Both use cases have been evaluated successfully, with 60% of testcases generated from the useful list of Z80 emulator, and 44% from the subset provided by ESAIL OBSW. It can be also improved by modifying the parameters of the tool, like the used templates.*

MOTIF *For simpler cases, where reaching the mutated code is independent of class state, the automatic test generation was found to be easy and straight-forward.*

NEGATIVE COMMENTS

- *Integration of MASS with the test frameworks was found to be difficult due to large number of dependencies of MASS.*
- **Action:** MASS configuration depends on the need for acquiring information about test cases (e.g., what are the program units that correspond to each test case, how to run them, and how to collect code coverage. Programming assistants based on large language models could be investigated in the future to reduce configuration time.
- *Moreover, it was found that the applicability of MASS to particular test suites can be limited by the large CPU time required by test execution.*
- **Action:** MASS execution time often depends on the characteristics of the mutation analysis (e.g., test cases need to be executed). A solution based on the selection of few mutants has been positively evaluated.
- *The application of SnT toolsets to real projects could be hard in some points.*
  - *The complexity of embedded systems is very high, so it would be necessary to optimize MASS to simplify the generation of mutants and detect mutants that are redundant or equivalent.*
  - **Action:** MASS already includes solution for redundant and equivalent mutants; however, they could be improved further with additional static analysis.
  - *Some of the results of the tests could be nondeterministic or depending on different factors like elapsed time or hardware status.*
  - **Action:** MASS now includes a solution to handle non-determinism.

DAMAT *In practice the manual effort to make suitable mutations and execute tests limits its (DAMAT) usability to selected critical functionalities.*

- **Action:** Further improvements of DAMAT will require research on the definition of fault models, which might be ahcieved through large language models or model inference.

MOTIF *Since S5 L1bPP SW is implemented with C++ and relies on classes with complex initialization and state, it was found that reaching the mutated code tends to require implementation of manual test drivers.*

- **Action:** The generation of drivers may benefit from the adption of large language models, which could be topic for future research activities.

MOTIF *since the generated test cases rely on data in byte arrays, they can become dependent on the execution platform and dependencies.*

- **Action:** One possible solution is to automate the translation of exadecimal inputs into human-readable ones, thus preventing such issues.

SUGGESTED IMPROVEMENTS

- MASS/MOTIF
  - *Support for more programming languages.*
  - *A graphic/text user interface to execute all the steps in an interactive and easy way.*
  - *Integration between both tools in a unified framework.*
  - *Generation of charts and statistics as outputs for both tools.*
- DAMAT
  - *A graphic interface to create the definitions of the mutation operators, with a description and examples for each of them.*
  - *Automatize the changes to be performed in the code to apply the defined mutations. It can be a very complicated task, so another option would be to suggest to the user different code templates to be used in the SUT.*
  - *Generation of fault models in an automatic way, according to several inputs defined by the user about the structure of the SUT.*
  - *Generation of test cases from live mutants with mutation operations defined in the fault model. It would be useful to generate these test cases by using an existing test framework (e.g., Google Test, Check)*
  - *Integration of the toolset with MASS and MOTIF. As seen in the use case for the Z80 emulator, both toolsets are complementary*
  - *Support for more programming languages when generating code from fault models, like Python or Rust.*

6 INTEGRATION WITH ISVV AND ECSS PRACTICES

6.1 Overview

According to the ECSS standard ECSS-E-ST-40C, key validation campaigns are *unit testing*, *integration testing*, and *validation with respect to requirements baselines*, which we refer to as system-level testing because it typically involves the execution of the whole software, either within a simulation environment or with target hardware. Acceptance testing occurs later in the process and is not part of the application context for this document. Figure 10 provides an overview of the different test levels specified by the ECSS standards (unit, integration, and system) and the software interactions stressed by them. 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.

Table 7 provides an overview of the applicability of the FAQAS-2 methodology to different testing levels. *Unit testing* is an ideal target for *code-driven mutation analysis*; indeed, test cases focus on

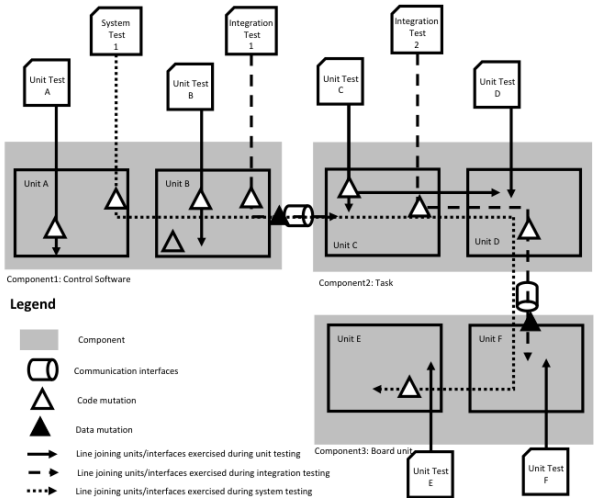

Fig. 10. Mutation Testing Approaches for Different Testing Levels.

Table 7. Support of code-driven and data-driven mutation analysis (assessment of test cases) and testing (generation of test cases), for different testing levels.

	Code-driven		**Data-driven**	
Testing level	**analysis** (MASS)	**testing** (MOTIF)	**analysis** (DAMAT)	**testing** (DAMTEF)
Unit	Yes	Yes	No	No
Integration	Yes	Yes	Yes	Yes
System	Yes	No	Yes	No

specific code portions and can be executed quickly, which speeds up mutation analysis. However, MASS, the code-driven mutation analysis tool of FAQAS-2, integrates solutions to make mutation analysis scale and renders it applicable in the context of *integration testing* and *system testing*.

Unit testing is also the target for *code-driven mutation testing* (i.e., test generation) in FAQAS-2; indeed, the MOTIF tool automatically generates unit test cases. MOTIF can also *generate integration test cases* that exercise multiple units when there is a function under test that acts as an entry point for exercising multiple units. Instead, the code-driven generation of *system test cases* is not supported by FAQAS-2, and, to the best of our knowledge, it's not supported by any other mutation testing approach in the literature.

Data-driven mutation techniques, instead, are *unlikely useful* in the context of *unit testing* because single units executed in isolation do not exchange messages (the target of data-driven mutation). Data-driven mutation analysis, implemented by DAMAT, *targets integration and system test cases* because they both trigger the exchange of messages. *Data-driven mutation testing*, which is implemented by DAMTEF, is feasible only for *integration test cases* because the fuzzing-based generation of system-level test cases implemented by DAMTEF is an open research problem not addressed by FAQAS-2.

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 of the mutants generated for the system. We therefore suggest *computing 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).

Since a key objective of test suite assessment is the ***determination of test suite adequacy***, FAQAS-2 led to the identification of guidelines enabling the assessment of test suite adequacy based on the mutation score, which shall suggest if further test cases are needed, both within nominal and ISVV context. Specifically, when performing ***code-driven mutation analysis*** to assess the quality of test suites, we suggest using the following values as reference values for the mutation score:

- (RQ3) 40%: A mutation score below 40% indicates a poor quality test suite that may not detect any fault.
- (RQ1) 70%: Assuming that the test suite maximizes structural coverage, a 70% mutation score enables improving fault detection significantly.
- (RQ2) 84%: Assuming that the test suite maximizes structural coverage, 84% mutation score is highly desirable, to maximize fault detection rate (e.g., for category A and B software).

When performing ***data-driven mutation*** analysis to assess the quality of test suites, we suggest using the following values as reference values for the mutation score:

- (RQ3) 20%: A mutation score below 20% indicates a poor quality test suite that may not detect any fault.
- (RQ1) 65%: Assuming that the test suite maximizes structural coverage, a 65% mutation score enables improving fault detection significantly.
- (RQ2) 90%: Assuming that the test suite maximises structural coverage, a 90% mutation score is highly desirable to maximize fault detection rate.

6.2 Matching with ECSS standards

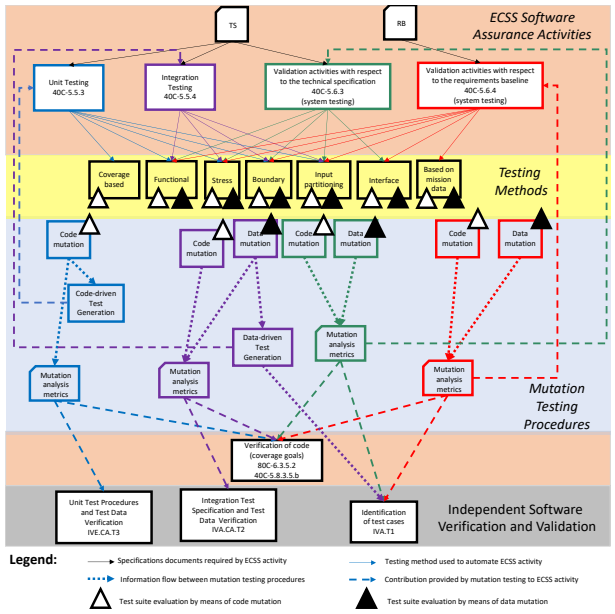

Fig. 11. Relations between ECSS activities and activities of the mutation testing process.

Figure 11 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, data-driven test generation, 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 11, 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 fuzzing can be used in the context of unit and integration testing targeted by code-driven mutation (with MOTIF) and in the context of integration testing targeted by data-driven mutation (with DAMTEF). System test suites can be improved only manually.

In Figure 11, 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) 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 [11] 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 [11]) and Integration Test Specification and Test Data Verification (IVA.CA.T2 in [11]). Finally, test generation and mutation score may support ISVV during the identification of test cases (IVA.T1 in [11]).

6.3 ISVV integration

The main limiting factor for the integration of mutation analysis in an ***ISVV context***, more precisely, what limits the possibility for an ISVV supplier to rely on mutation analysis/testing for test suite assessment, is the need for executing validation test cases (unit or integration ones).

Indeed, the execution of the validation tests is possible only in a subset of ISVV projects. More commonly, ISVV suppliers either execute tests at customer premises or witness the test execution by the customer. Further, unit tests are not usually part of ISVV projects, which complicates the applicability of mutation analysis to ISVV. Finally, flight software validation tests often require simulation of the target platform, which makes the execution of tests significantly slower, and the number of tests can be very large.

However, for ***code-driven mutation analysis***, it is deemed feasible for developers to execute mutation analysis on their premises and provide results (mutation score and source code of live

mutants) to ISVV providers, who can give feedback based on the mutation score or live mutants. The suggested procedure, experimented with ExoMars, consists of the following steps:

- The ISVV supplier compiles mutants and keeps the ones being neither equivalent nor redundant based on trivial compiler optimizations
- The ISVV supplier assigns cyclomatic complexity (*CC*) to each mutant, following the procedure in use by MASS.
- The ISVV supplier assigns to each mutant the number of bug reports mentioning either the function or the module (identified as the file) from which the mutant was generated. We will call this value *MR*.
- The ISVV supplier scores mutants according to a formula taking into account CC and MR, and sorts mutants based on the computed score.
- The ISVV supplier selects the top N mutants by maximizing diversity.

The integration of ***data-driven mutation analysis*** into ISVV practices is considered more difficult because of the necessity for the ISVV provider to acquire a complete understanding of the data model to derive a fault model, and to identify what test cases should be expected to fail in case of data-driven mutation. One possible solution is to expect the software supplier to draft a fault model, fo each mutant, execute at least one test cases that is supposed to fail and provide results to the ISVV supplier.

7 TOOLSET LIMITATIONS AND OPEN PROBLEMS

FAQAS-2 led to developing tools that enable the application of mutation analysis and testing to space software. FAQAS-2 extended the developments done in FAQAS-1, ensuring the generalizability and applicability of the tools that achieved the highest TRLs in FAQAS-1 (i.e., MASS and MOTIF), and building on the experience gained with test generation tools (i.e., SEMuS and DAMTE) to develop new tools (i.e., MOTIF and DAMTEF) that substantially overcome the limitations of previous ones thanks to the adoption of a fuzzing approach. MOTIF enables dealing with programs with floating-point variables, can generate test cases that trigger the execution of units communicating with channels, and outperform the previous tools in term of percentage of killed mutants. DAMTEF supports a broader set of architectures and improves the quality of results.

However, the FAQAS-2 toolset remain affected by a few litimitations; some concern the usability of the mutation analysis tools, which reached the highest TRLs, others concern the generalizability of the test generation approaches. We describe current limitations in the following paragraphs.

MASS and DAMAT reached TRL6; however, their usability remains affected by some pitfalls:

- MASS still requires the manual production of some configuration files, which might be discouraging for software engineers.
- MASS requires the testing of the system multiple times, which may lead to a long mutation analysis phase.
- DAMAT requires the manual definition of a fault model, which requires effort and might be discouraging.
- Both MASS and DAMAT do not provide safety guarantees in case of application with hardware in the loop (e.g., they do not ensure preventing hardware damages caused by code mutation).
- MASS and DAMAT target C and C++ software, which are the main languages for flight and ground software. However, other languages such as Java and Rust are becoming popular in the space sector and might need to be supported for the application of mutation analysis and testing. Code-driven mutation analysis and testing tools exist for Java, and FAQAS-2 demonstrated the feasibility of their adoption; however, the FAQAS-2 developments did not

lead to high TRLs for Java mutation analysis and testing. Further, FAQAS-2 has shown the feasibility of applying DAMAT to Java software, although only a low TRL demonstrator has been developed.

MOTIF reached TRL4; however, its applicability remains affected by the following issues:

- With C software, MOTIF cannot test functions that work with global variables.
- With C software, MOTIF cannot test functions that receive struct with pointers.
- With C++ software, MOTIF requires manual intervention to test private and protected methods.
- With C++ software, MOTIF requires manual intervention to instantiate classes whose constructors require objects as input.
- With C++ software, MOTIF requires manual intervention to test abstract and template methods; for the former, the end-user selects which concrete subclass to instantiate, and for the latter, the end-user selects which class to provide in the template.
- With both C and C++, MOTIF relies on binary values to initialise input variables and test assertions, which reduces readability and prevents cross-compilation (because of endianness).
- MOTIF enables killing the live mutants created by MASS. Since MASS injects mutants into statements covered by the test cases, MOTIF cannot generate test cases for statements not exercised by test cases, although this might be a desirable feature because test suites often do not reach adequate statement coverage.

DAMTEF reached TRL 2, and is affected by the following limitations:

- the identification of the interfaces to be exercised to drive the software under test towards producing the data messages to be mutated by DAMAT requires a complex integration of static and dynamic information and has been therefore left to manual intervention;
- a wider set of case study subjects might be considered to ensure that the solution implemented in DAMTEF generalises.

7.1 Future developments

To improve the limitations above, a follow-on activity shall target the following objectives:

(1) To reduce the manual configuration effort required by MASS, MASS shall be integrated with development environments commonly used in space contexts. However, this would imply determining what is the common IDE choice for space projects development, which so far has not been possible because subcontractors highlighted that it seems to change from project to project. An alternative solution may consist of studying the feasibility of relying on large language models to determine project configurations automatically, instead of implementing parser specifics for different IDEs.
(2) To reduce MASS execution time, since MASS is best executed with several parallel cores, it would be useful to integrate it with Cloud provider APIs (e.g., Amazon AWS), to enable parallel execution natively.
(3) FAQAS-2 demonstrated that sorting MASS live mutants based on cyclomatic complexity increases the chances of detecting faults, which may suggest that the prioritisation of mutants based on cyclomatic complexity may help reducing mutation analysis effort (e.g., testing only mutants in functions with high cyclomatic complexity).
(4) For both MASS and DAMAT, their application with hardware in the loop might require the automated identification of variables impacted by the changes (e.g., through forward impact analysis) to eliminate mutants that affect critical registries or variables.

(5) Consolidate the Java mutation analysis and testing framework assessed in FAQAS-2 into a TRL6 toolset. It shall support widely adopted Java frameworks such as Spring[3].
(6) Consolidate the DAMAT support for Java ensuring integration with frameworks such as Spring.
(7) To simplify the definition of DAMAT fault models, the development of a question/answer system (e.g., chatbot), eventually supported by a large language model, might help.
(8) For MOTIF, an improved parser shall be implemented to deal with global variables. This parser will identify global variables to be used as inputs and verified as oracles.
(9) For MOTIF, the automated initialisation of complex data structures with pointers and complex objects can be achieved with improved static analysis or by leveraging solutions that build on large language models for similar purposes (e.g., the generation of drivers for fuzz testing).
(10) For MOTIF to test private and protected methods, it is necessary to improve the static analysis component. Specifically, it shall automatically determine what public methods use the private and protected functions under test. Those methods shall be used as entry points for the fuzzing procedure.
(11) For MOTIF, binary values in test cases shall be replaced with appropriate "readable" values, which might be obtained with an improved parser leveraging dynamic analysis or a large language model.
(12) MOTIF might be extended to preserve all the test inputs that enable reaching statements not exercised by the test suite, in addition to input that kills mutants. To achieve such an objective, it might be sufficient to introduce the control logic to determine coverage improvement and store all the inputs that achieve such coverage. Alternatively, MOTIF can be applied to all the mutants, including the ones concerning lines of code not covered by test cases. The current MOTIF components for test case generation shall be sufficient to transform the inputs identified in either case into test cases.
(13) For DAMTEF, application to a larger set of case study subjects is needed to determine if results (i.e., the feasibility of automatically generating test cases) generalise or if additional strategies are needed.

To achieve the improvements above, a follow-on activity would be needed. The activity shall lead TRL 6 for MOTIF and TRL 7 for MASS and DAMAT; instead, DAMTEF shall be improved to achieve TRL 4. The resources required for such an endeavour shall be comparable to the resources that were required for FAQAS-2.

[3] https://spring.io/

8 CONCLUSION

This report summarized the results of FAQAS-2. The activies concerned the imrpovement of the FAQAS-1 methodology to facilitate applicability to a larger range of case study subjects and facilitate automated test generation. For code-driven mutation analysis, FAQAS-2 led to extension of MASS to deal with the peculiarities of new case studies, the identification of a solution for the prioritization of mutants, the identification of thresholds for mutation analysis that can be used for test adequacy determination, and the definition of a mutation testing process for Java. For code-driven mutation testing, FAQAS-2 led to the development of MOTIF, an effective solution for the automatic generation of test cases that kill mutants, based on fuzzing. For data-driven mutation analysis, FAQAS-2 led to the extension of DAMAT to be applied to additional subjects, including Java systems. Finally, we investigated the feasibility of DAMTEF , which relies on fuzzing to generate test cases to kill data-driven mutants. Our results confirm the feasibility of all the proposed approaches.

REFERENCES

[1] M. Boehme, C. Cadar, and A. Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 03 (may 2021), 79–86. https://doi.org/10.1109/MS.2020.3016773

[2] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. 2006. Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Comput. Surv.* 38, 4 (Dec. 2006), 12?es. https://doi.org/10.1145/1177352.1177354

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[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] CLANG. 2022. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[6] 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

[7] GNU compiler collection. 2020. gcov?a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[8] 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.

[9] European Cooperation for Space Standardization. 2009. ECSS-E-ST-40C ? Software general requirements. http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/

[10] 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/

[11] European Space Agency. [n. d.]. ESA ISVV Guide issue 2.0, 29/12/2008.

[12] Free Software Foundation. 2020. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/

[13] 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.

[14] Marinos 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

[15] Jaekwon Lee, Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, and Lionel Briand. 2023. Fuzzing for CPS Mutation Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1389. https://doi.org/10.1109/ASE56229.2023.00079

[16] Jaekwon Lee, Enrico Viganò, Fabrizio Pastore, and Lionel Briand. 2024. MOTIF: A tool for Mutation Testing with Fuzzing. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 451–453. https://doi.org/10.1109/ICST60714.2024.00052

[17] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (nov 2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[18] 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

[19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[20] 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.

[21] Peach Tech. [n. d.]. Peach Fuzzer. https://www.peach.tech

[22] Thanassis Tsiodras. 2020. Cover me! https://www.thanassis.space/coverage.html

[23] E. Vigano, O. Cornejo, F. Pastore, and L. C. Briand. 2023. Data-Driven Mutation Analysis for Cyber-Physical Systems. *IEEE Transactions on Software Engineering* 49, 04 (apr 2023), 2182–2201. https://doi.org/10.1109/TSE.2022.3213041

[24] Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, and Lionel Briand. 2023. DaMAT: A Data-driven Mutation Analysis Tool. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 165–169. https://doi.org/10.1109/ICSE-Companion58688.2023.00047

[25] 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.