Applicability of Mutation Testing Method for Flight Software: Fault-based, Automated Quality Assurance Assessment for Space Software (FAQAS)

ENRICO VIGANÒ, OSCAR CORNEJO, FABRIZIO PASTORE, University of Luxembourg, Luxembourg

Abstract. This document is the final report of the ESA activity *ITT-1-9873-ESA (Applicability of Mutation Testing Method for Flight Software)*, which concerns the development of a framework for the automated assessment and the automated improvement of test suites for embedded software deployed on spacecrafts.

The activity led to the development of a toolset, the *FAQAS toolset*, which includes three tools *MASS*, *SEMuS*, and *DAMAt*. MASS (Mutation Analysis for Space Software) is a tool for the assessment of test suites based on mutation analysis. Mutation analysis evaluates the quality of a test suite by generating faulty software versions called mutants and by reporting the percentage of mutants detected by the test suite. MASS scales to large software systems because it relies on mutants sampling based on confidence interval estimation. SEMuS (Symbolic Execution Mutation testing for Space software) is a tool for the automated improvement of test suites; it automatically generates unit test cases that detect the presence of mutants. DAMAt (DAta-driven Mutation Analysis with Tables) is a tool that assesses the quality of test suites by simulating errors in the data exchanged by software components, different from MASS it simulates faults concerning components interoperability.

The *scalability* and *effectiveness* of the FAQAS toolset has been demonstrated through the application of the toolset to case study systems provided by GomSpace, LuxSpace, and ESA. Both MASS and SEMuS enabled the identification of faults affecting the software under analysis. Both MASS and DAMAt enabled the identification of major pitfalls in the test suite. Mutation analysis has been demonstrated to be feasible (i.e., executable in few days even for large systems); however, adequate computational resources (e.g., multiple computation nodes) are necessary. The automated generation of unit test cases, instead, can produce useful results in few minutes. Our results show that the FAQAS toolset enables ensuring high-quality in space software.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Mutation analysis, CPS, CPS Interoperability, Integration testing

1 INTRODUCTION

From spacecrafts to ground stations, software has a prominent role in space systems; for this reason, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Mission failures due to insufficient software sanity checks [12] are unfortunate examples, pointing to the necessity for systematic and predictable quality assurance procedures in space software.

Existing standards for the development of space software regulate software quality assurance and emphasize its importance. The most stringent regulations are the ones that concern flight software, i.e., embedded software installed on spacecrafts, our target in this activity. 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 the definition and assessment of test suites [8, 10].

Test suites assessment is typically based on code inspections performed by space authorities and independent software validation and verification (ISVV) activities, which include the verification of test procedures and data (e.g., ensure that all the requirements have been tested and that representative input partitions have been covered [11]). Though performed by specialized teams, such assessment is manual and thus error prone and time-consuming. *Automated and effective methods*

to evaluate the quality of the test suites are thus necessary. Also, *methods to automatically generate test cases will speed-up the improvement of test suites.*

Since one of the primary objectives of software testing is to identify the presence of software faults, an effective way to assess the quality of a test suite consists of artificially injecting faults in the software under test and verifying the extent to which the test suite can detect them. This approach is known as *mutation analysis* [6]. In mutation analysis, faults are automatically injected in the program through automated procedures referred to as mutation operators. Mutation operators enable the generation of faulty software versions that are referred to as *mutants*. Mutation analysis helps evaluate the effectiveness of a test suite, for a specific software system, based on its mutation score, which is the percentage of mutants leading to test failures.

Despite its potential, mutation analysis is not widely adopted by industry in general and space system development in particular. The main reasons include its limited scalability and the pertinence of the mutation score as an adequacy criterion [25]. Indeed, for a large software system, the number of generated mutants might prevent the execution of the test suite against all the mutated versions. Also, the generated mutants might be either semantically equivalent to the original software [22] or redundant with each other [29]. Equivalent and redundant mutants may bias the mutation score as an adequacy criterion.

The mutation analysis literature has proposed several optimizations to address problems related to scalability and mutation score pertinence. For example, scalability problems are addressed by approaches that sample mutants [31] [16] or solutions that prioritize and select the test cases to be executed for each mutant [32]. Equivalent and redundant mutants can be detected by comparing the code coverage of the original program and its mutants [17, 26–28]. However, these approaches have not been evaluated on industrial, embedded systems and there are no feasibility studies concerning the integration of such optimizations and their resulting, combined benefits. For example, we lack mutants sampling solutions that accurately estimate the mutation score in the presence of reduced test suites; indeed, mutant sampling, which comes with a certain degree of inaccuracy, may lead to inaccurate results when applied to reduced test suites that do not have the same effectiveness of the original test suite.

In addition, existing mutation analysis approaches cannot identify problems related to the interoperability of integrated components (integration testing). Space software, similar to software running in other types of CPSs, is often affected by problems cause by the lack of *interoperability of integrated components* [15, 19], mainly due to the wide variety and heterogeneity of the technologies and standards adopted. It is thus of fundamental importance to ensure the effectiveness of test suites with respect to detecting interoperability issues, for example by making sure test cases trigger the exchange of all possible data items and report failures when erroneous data is being exchanged by software components. For example, the test suite for the control software of a satellite shall identify failures due to components working with different measurement systems [23]. Unfortunately, well known, code-driven mutation operators (e.g., the sufficient set [4, 5]) simulate algorithmic faults by introducing small changes into the source code and are thus unlikely to simulate interoperability problems resulting in exchanges of erroneous data.

Finally, traditional mutation analysis approaches *cannot inject faults into black-box components* whose implementation is not tested within the development environment (e.g., because it is simulated or executed on the target hardware). For example, in a satellite system, such components include the control software of the Attitude Determination And Control System (ADCS), the GPS, and the Payload Data Handling Unit (PDHU). During testing with simulators in the loop, the results generated by such components (e.g., the GPS position) are produced by a simulator. During testing with hardware in the loop, these components are directly executed on the target hardware

and cannot be mutated, either because they are off-the-shelf components or to avoid damages potentially introduced by the mutation.

Last, test generation approaches are in preliminary stages and cannot be applied in industrial space context. For example, SEMU, a state-of-the-art approach can generate test inputs only for batch programs that can be compiled with the LLVM infrastructure.

FAQAS had the objective to assess the feasibility of mutation analysis and testing for space software by identifying feasible solutions based on existing literature. FAQAS objectives were the following:

- To perform a comprehensive analysis and survey of mutation analysis/testing.
- To prototype the mutation analysis/testing process to be applied on space software.
- To develop a toolset supporting mutation analysis and testing automation.
- To empirically evaluate mutation analysis/testing by applying it to space software use cases.
- To evaluate how mutation analysis/testing can be integrated into a typical verification and validation life cycle of space software and to define a mutation analysis/testing methodology.

Overview of the contributions

FAQAS led to the development of a toolset that includes the following tools:

- MASS (Mutation Analysis for Space Software), a tool that automatically executes code-driven mutation analysis. Code-driven mutation analysis consists of automatically generating mutants by altering the source code of the software under test. MASS implements a pipeline that makes it feasible in the context of space software.
- DAMAt (DAta-driven Mutation Analysis with Tables), a tool that automatically executes data-driven mutation analysis. Data-driven mutation analysis is an approach newly defined within FAQAS, which, instead of mutating the implementation of the software under test, alters the data exchanged by software components. 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.
- SEMuS (Symbolic Execution-based MUtant analysis for Space software), a tool that automatically generates test inputs based on code-driven mutation analysis results.

The activity also included an extensive empirical evaluation demonstrating the feasibility, effectiveness, and scalability of the proposed approaches with space software. Finally, the activity evaluated the feasibility of DAMTE, a tool-supported methodology for the generation of test inputs based on data-driven mutation analysis results.

Figure 1 provides an overview of the input and outputs of the FAQAS toolset. All the components take as input the software under test (SUT), its test suite, and a set of configuration files. MASS generates as output a set of live mutants (i.e., mutants that do not lead to a test suite failure), a set of killed mutants (i.e., mutants that do not lead to a test suite failure), and information useful to draft a verification report, which includes the statement coverage of the SUT test suite, the mutation score, and the execution time of test cases. SEMuS takes as input the list of live mutants detected by MASS and aims to generate test cases that kill them. After the execution of SEMuS, engineers have a set of additional test cases to be integrated into the SUT test suite and a list of live mutants (i.e., mutants for which SEMuS was not able to identify test inputs killing them). Live mutants shall be manually inspected by engineers to either determine if they are equivalent mutants or to manually derive a test input killing them. DAMAt generates as output a set of killed mutants (i.e., mutants that, during testing, successfully alter the data, and lead to test case failures), a set of live mutants (i.e., mutants that, during testing, successfully alter the data, but do not lead to test case failures), and a set of not executed mutants (i.e., mutants that, during testing, could not alter any data because

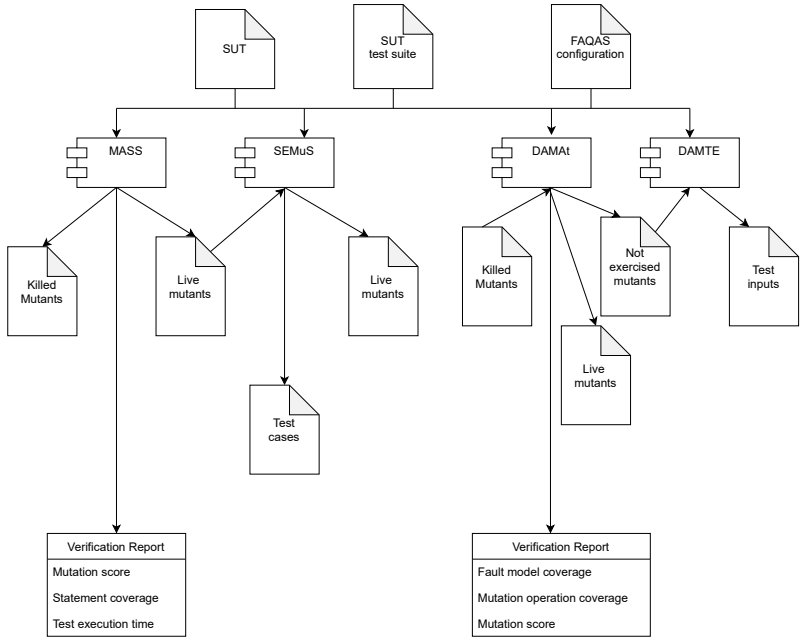

Fig. 1. Overview of the FAQAS toolset

the data they target is never exercised by the SUT); also, it provides information useful to draft a verification report, which includes the fault model coverage, the mutation operation coverage, and the mutation score. DAMTE is a manual procedure supported by the KLEE toolset that enables an engineer to automatically derive inputs that increase the fault model coverage and the mutation operation coverage.

Sections 2 to 5 provide an overview of the FAQAS tools: MASS, SEMuS, DAMAt, and DAMTE. Section 6 introduces the case study subjects considered for empirical evaluation. Section 7 provides an overview of the empirical results obtained. Section 8 concludes this report.

2 CODE-DRIVEN MUTATION ANALYSIS: MASS

2.1 Overview

Figure 2 provides an overview of the mutation analysis process that we propose, *Mutation Analysis for Space Software (*DaMAT*)*. Its goal is to propose a comprehensive solution for making mutation analysis applicable to embedded software in industrial cyber-physical systems. The ultimate goal of *DaMAT* is to assess the effectiveness of test suites with respect to detecting violations of functional requirements.

DaMAT consists of eight steps: (Step 1) Collect SUT Test Suite Data, (Step 2) Create Mutants, (Step 3) Compile Mutants, (Step 4) Remove Equivalent and Duplicate Mutants Based on Compiled Code, (Step 5) Sample Mutants, (Step 6) Execute Prioritized Subset of Test Cases, (Step 7) Identify Likely Equivalent / Duplicate mutants Based on Coverage, and (Step 8) Compute the Mutation Score. Different from related work, *DaMAT* enables FSCI-based sampling by iterating between

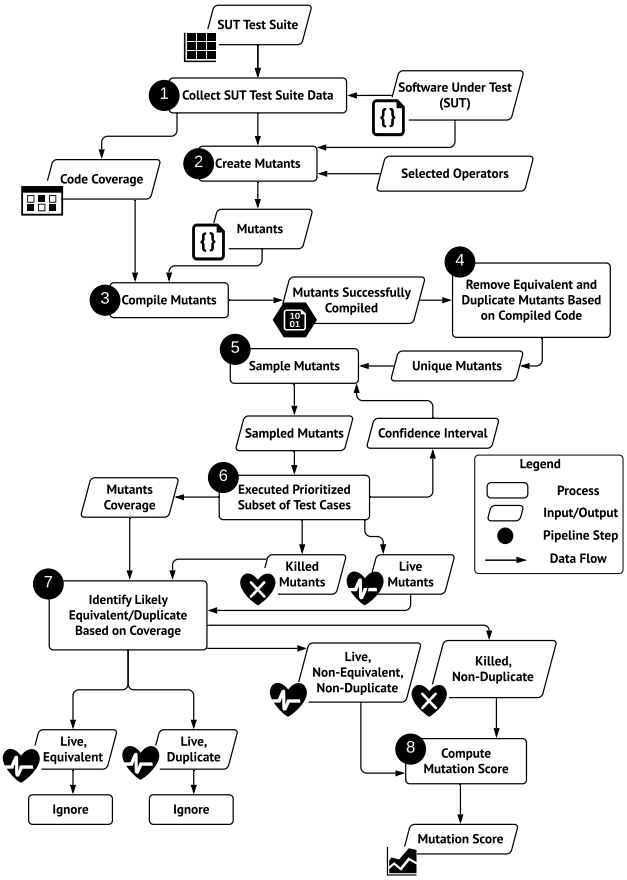

Fig. 2. Overview of MASS

mutants sampling (Step 5) and test cases execution (Step 6). Also, it integrates test suite prioritization and reduction (Step 6) before the computation of the mutation score. Finally, it includes methods to identify likely equivalent and duplicate mutants based on code coverage (Step 7). We describe each step in the following paragraphs.

Table 1. Implemented set of mutation operators.

	Operator	Description*
Sufficient Set	ABS	$\{(v, -v)\}$
	AOR	$\{(\mathrm{op}_1, op_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{+, -, *, /, \%\} \wedge \mathrm{op}_1 \neq \mathrm{op}_2\}$ $\{(\mathrm{op}_1, \mathrm{op}_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{+=, -=, *=, /=, \%=\} \wedge \mathrm{op}_1 \neq \mathrm{op}_2\}$
	ICR	$\{i, x) \mid x \in \{1, -1, 0, i+1, i-1, -i\}\}$
	LCR	$\{(\mathrm{op}_1, \mathrm{op}_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{\&\&, \|\|\} \wedge \mathrm{op}_1 \neq \mathrm{op}_2\}$ $\{(\mathrm{op}_1, \mathrm{op}_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{\&=, \|=, \&=\} \wedge \mathrm{op}_1 \neq \mathrm{op}_2\}$ $\{(\mathrm{op}_1, \mathrm{op}_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{\&, \|, \&\&\} \wedge \mathrm{op}_1 \neq \mathrm{op}_2\}$
	ROR	$\{(\mathrm{op}_1, \mathrm{op}_2) \mid \mathrm{op}_1, \mathrm{op}_2 \in \{>, >=, <, <=, ==, !=\}\}$ $\{(e, !(e)) \mid e \in \{\mathrm{if(e)}, \mathrm{while(e)}\}\}$
	SDL	$\{(s, \mathrm{remove}(s))\}$
	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	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{», «\}\}$
Other	LVR	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0),$ $(\mathrm{true}, \mathrm{false}), (\mathrm{false}, \mathrm{true})\}\}$

*Each pair in parenthesis shows how a program element is modified by the mutation operator on the left; we follow standard syntax [21]. 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).

2.2 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 [2] and GDB [13], enabling the collection of coverage information for embedded systems without a file system [30].

2.3 Step 2: Create Mutants

In Step 2, we automatically generate mutants for the SUT by relying on a set of selected mutation operators. In *DaMAT*, we rely on an extended sufficient set of mutation operators, which are listed in Table 2.

2.4 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. Mutants that lead to compilation errors are discarded.

2.5 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. After each execution of the compiler, we compute the SHA-512 hash summary of the generated executable.To detect equivalent mutants, *DaMAT* compares the hash summaries of the mutants with that of the original executable. To detect duplicate mutants but avoid combinatorial explosion, *DaMAT* focuses its comparison of hash summaries on pairs of mutants belonging to the same source file (restricting the scope of the comparison is common practice [20]). Hash comparison allows us to (1) determine the presence of equivalent mutants (i.e., mutants having the same

hash as the original executable), and (2) identify duplicate mutants (i.e., mutants with the same hash). Equivalent and duplicate mutants are then discarded.

2.6 Step 5: Sample Mutants

In Step 5, *DaMAT* samples the mutants to be executed to compute the mutation score. *DaMAT* does not selectively generate mutants but samples them from the whole set of successfully compiled, nonequivalent, and nonduplicated mutants (result of Steps 2 to 4). This choice aims to avoid sampling bias which may result from the presence of such mutants; indeed, there is no guarantee that these mutants, if they were discarded after being sampled, would be uniformly distributed across program statements. Our choice does not affect the feasibility of *DaMAT* since Steps 2 to 4 have negligible cost.

Our pipeline supports different sampling strategies: *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*, which is an innovative contribution of the FAQAS actvity [3].

2.7 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. Similarly to the approach of Zhang et al. [32], we define the order of execution of test cases based on their estimated likelihood of killing a mutant. However, in our work, this likelihood is estimated differently since, as discussed above, the measurements they rely on are not applicable in the context of system-level testing and complex cyber-physical systems. In contrast, to minimize the impact of measurements on real-time constraints, we only collect code coverage information for a small part of the system.

We execute only covered statements assuming that the test suite is optimal with respect to code coverage. More precisely, we addume that if a statement is not covered there is a good reason for it (e.g., it depends on hardware). If a statement is not covered by the test suite, there is no chance that a mutant generated in the non-covered statement can be possibly detected by any test case. If the test suite does not reach the required coverage there is no reason to perform mutation testing, because is already known that the test suite is not good.

2.8 Step 7: Discard Mutants

In this step, we identify likely nonequivalent and likely nonduplicate mutants by relying on code coverage information collected in the previous step.

Similarly to related work [28], we identify nonequivalent and nonduplicate mutants based on a threshold.

In our case, consistently with previous steps of *DaMAT*, we compute normalized distances based on the distance metrics D_J, D_O, D_E, and D_C. A mutant is considered nonequivalent when the distance from the original program is above the threshold T_E, for at least one test case. Similarly, a mutant is considered nonduplicate when the distance from every other mutant is above the threshold T_D, for at least one test case. For the identification of nonequivalent mutants, we consider live mutants only. To identify nonduplicate mutants, we consider both live and killed mutants; however, to avoid combinatorial explosion, we compare only mutants belonging to the same source file (indeed, mutants belonging to different files are unlikely to be redundant). Killed mutants that lead to the failure of different test cases are not duplicate, regardless of their distance.

Thresholds T_E and T_D should enable the identification of mutants that are guaranteed to be nonequivalent and nonduplicate. In particular, we are interested in the set of *live, nonequivalent, nonduplicate mutants* (hereafter, LNEND) and the set of *killed, nonduplicate mutants* (hereafter,

KND). With such guarantees, the mutation score can be adopted as an adequacy criterion in safety certification processes. For example, certification agencies may require safety-critical software to reach a mutation score of 100%, which is feasible in the presence of nonequivalent mutants.

2.9 Step 8: Compute Mutation Score

The *mutation score* (MS) is computed as the percentage of killed nonduplicate mutants over the number of nonequivalent, nonduplicate mutants identified in Step 7):

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

3 CODE-DRIVEN MUTATION TESTING: SEMUS

3.1 Overview

To achieve *test suite augmentation* (i.e., automatically generate test cases that kill mutants), in FAQAS, we have implemented an extension of SEMu that we call SEMu for Space Software (*SEMuS*).

In the FAQAS context, we cannot use SEMu as it is, since it requires mutants to be compiled with the *LLVM compiler* in LLVM-IR format. Any analysis requiring compilation into LLVM format is unlikely applicable to space software because of two reasons: (1) case studies often rely on compiler pipelines (e.g., RTEMS) that include architecture-specific optimizations that are not supported by LLVM, and (2) there is no guarantee that the compiled objects produced by LLVM are equivalent to those produced by the original compiler (e.g., memory allocation).

To overcome the limitations above and *apply the test generation approach of SEMu in FAQAS* we have implemented three solutions. First, to avoid mutants to behave differently than the original software because of LLVM specificities (case 2 above), we rely on MASS to identify killed and live mutants, and then compile only the live mutants into LLVM-IR format. Second, to increase the likelihood of successfully compiling the SUT with LLVM (case 1 above), instead of compiling the whole software, we compile only the mutated function and its dependencies, which enables the generation of unit test cases and shall be sufficient to ensure the quality of the system under test in this context (e.g., unit test cases are normally used to ensure high code coverage). Third, to enable the generation of the meta-mutants, which is necessary to apply SEMu, we have extended MASS to (1) generate, for each source file of the SUT, both a meta-mutant processed by SEMu and the mutants processed by MASS, and (2) trace each mutant, after mutation analysis, to each mutant contained into the meta-mutant.

Figure 3 shows the architecture of SEMuS and how it interacts with MASS. SEMuS consists of five components, which are *Test Template Generator*, *Pre-SEMu*, *KLEE-SEMu*, *KTest to Unit Test*, and *LLVM*. They are detailed in the following paragraphs.

3.2 Test Template Generator

The *Test Template Generator* (TTG) component automates the generation of templates for the symbolic execution search. The component receives as inputs the SUT source code and the list of SUT functions.

Listing 1 shows an example of a test template generated by the TTG. The TTG generates a template for every SUT function. The TTG parses the function arguments and declares them symbolic through use of the KLEE function `klee_make_symbolic`. Then, it adds a call to the function under analysis with symbolic values, and it saves the return value into a support variable (i.e., `result_faqas_semu` in Listing 1). Finally, it generates a number of invocations of the *printf* function

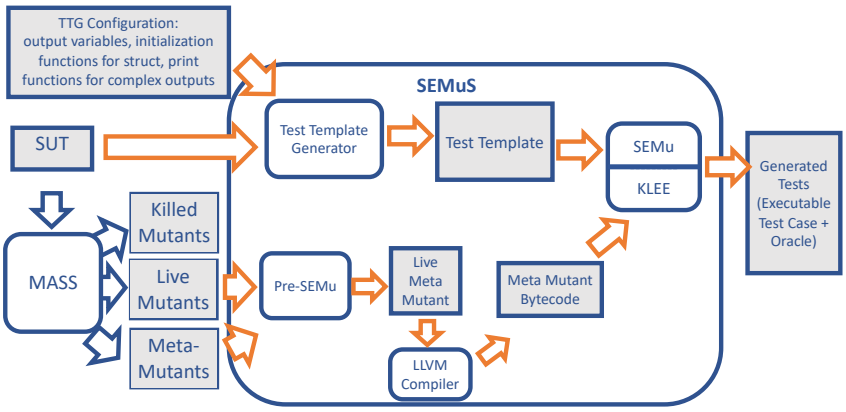

Fig. 3. FAQAS-SEMuS Architecture and Workflow

that print the value of the software outputs and adds a return statement with the value returned by the function under test (e.g., result_faqas_semu in Listing 1).

```
int main(int argc, char** argv) {
    // Declare variable to hold function returned value
    _Bool result_faqas_semu;
    // Declare arguments and make input ones symbolic
    unsigned long pVal;
    int pErrCode;
    klee_make_symbolic(&pVal, sizeof(pVal), "pVal");
    // Call function under test
    result_faqas_semu = T_INT_IsConstraintValid(&pVal, &pErrCode);
    // Make some output
    printf("FAQAS-SEMU-TEST_OUTPUT: %d\n", pErrCode);
    printf("FAQAS-SEMU-TEST_OUTPUT: %d\n", result_faqas_semu);
    return (int)result_faqas_semu;
}
```

Listing 1. SEMuS test template.

3.3 Pre-SEMu

The *Pre-SEMu* component generates *mutant schemata*; specifically, the component includes and compiles all the live mutants (i.e., MASS output) into a single bytecode file named the *Meta Mutant*. SEMu will select which mutant to consider for test generation based on a parameter. The compilation of the Meta Mutant into LLVM bitcode is enabled by the *LLVM* compiler infrastructure.

3.4 KLEE-SEMu

KLEE-SEMu is the underlying test generation component. This component receives as inputs the *LLVM bitcode* of the *Meta Mutant* and the *Test Template* for the function under test, and proceeds to apply dynamic symbolic execution to generate test inputs to kill the mutants. The output of this component are the *KLEE tests*.

A *KLEE test* is a binary file that contains information about the execution of KLEE such as the entry point of the analysis, and the generated test inputs.

```c
#include <stdio.h>
#include <string.h>

#include "asn1crt.c"
#include "asn1crt_encoding.c"
#include "asn1crt_encoding_uper.c"


int main(int argc, char** argv)
{
    (void)argc;
    (void)argv;

    // Declare variable to hold function returned value
    _Bool result_faqas_semu;

    // Declare arguments and make input ones symbolic
    unsigned long pVal;
    int pErrCode;
    memset(&pVal, 0, sizeof(pVal));
    const unsigned char pVal_faqas_semu_test_data[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
     x00, 0x00};
    memcpy(&pVal, pVal_faqas_semu_test_data, sizeof(pVal)); // Unsigned val is 0

    // Call function under test
    result_faqas_semu = T_INT_IsConstraintValid(&pVal, &pErrCode);

    // Make some output
    printf("FAQAS-SEMU-TEST_OUTPUT: pErrCode = %d\n", pErrCode);
    printf("FAQAS-SEMU-TEST_OUTPUT: result_faqas_semu = %d\n", result_faqas_semu);
    return (int)result_faqas_semu;
}
```

Listing 2. Generated test case

3.5 KTest to Unit Test

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

Listing 2 shows an example of a test case generated for a mutant present in the function T_INT_IsConstraintValid.

3.6 Test suite augmentation

The procedure for testing with SEMuS is shown in Figure 4. In Step 1, the engineer executes SEMuS, which generates an output folder for every live mutant that is killed by the generated test cases. Every folder contains: a script (i.e., *runTest.sh*) that can be used to execute the generated test case, (2) the test case itself (i.e., test1.c), and (3) a text file with extension *.expected* that contains the output that is observed when executing the test case with the SUT. In Step 2, the engineer visually inspects every file with extension *.expected* to determine if the observed output matches the specifications; if not, the software is faulty and needs to be fixed. In this case mutation testing enables detecting a fault. After verifying all the generated files with extension *.expected* the engineer can reuse the test cases generated by SEMuS for regression testing in future versions of the SUT. Basically, the output folders generated by SEMuS become part of the test suite of the SUT.

When there is a new version of the SUT (Step 3, in Figure 4), the folder with the source code of the SUT is replaced with the new version of the SUT (this can be done automatically with version control software). The engineer can then trigger test execution by simply re-executing

all the scripts *runTest.sh* generated by SEMuS. The script *runTest.sh* first executes the test case (Step 4.1), then it stores the test outputs into a text file with extension *.got*, finally if compares the observed output with the output generated for the previous version. If the function under test was not modified, differences may indicate that the test case FAILED.

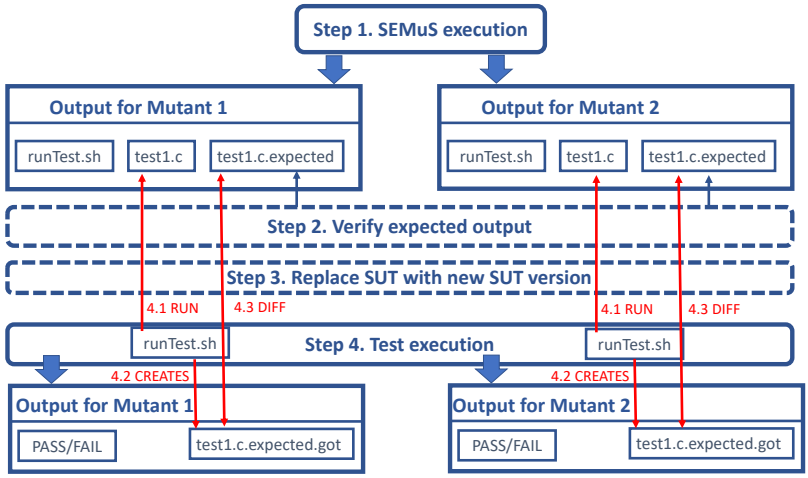

Fig. 4. Workflow for test suite augmentation with SEMuS

4 DATA-DRIVEN MUTATION ANALYSIS: DAMAT

Data-driven mutation analysis is a new mutation analysis paradigm that alters the data exchanged by software components to evaluate the capability of a test suite to detect interoperability faults. 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 are 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 are 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.

Although different types of fault models might be envisioned, we propose a technique (*data-driven mutation analysis with tables*, DAMAt), which automates data-driven mutation analysis by relying on a tabular block model, itself tailored to the SUT through predefined mutation operators. To concretely perform data mutation at runtime, DAMAt relies on a set of *mutation probes* that shall be integrated by software engineers into the software layer that handles the communication between components. The runtime behaviour of mutation probes (i.e, what data shall be mutated and how) is driven by the fault model. Thus, DAMAt can automatically generate the implementation of mutation probes from the provided fault model. Depending on the CPS, probes might be inserted either into the SUT, into the simulator infrastructure, or both. For example, Figure 5 shows the architecture of the ESAIL satellite system with mutation probes integrated into the SVF functions that handle communication with external components (PDHU, GPS, and ADCS in this case).

DAMAt works in six steps, which are shown in Figure 6. In Step 1, based on the provided methodology and predefined mutation operators, the engineer prepares a fault model specification tailored to the SUT. In Step 2, DAMAt generates a mutation API with the functions that modify the data according to the provided fault model. 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). 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 mutation operation (i.e., a mutation procedure configured for a data item, according to our terminology). 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. In Step 6, DAMAt generates mutation analysis results.

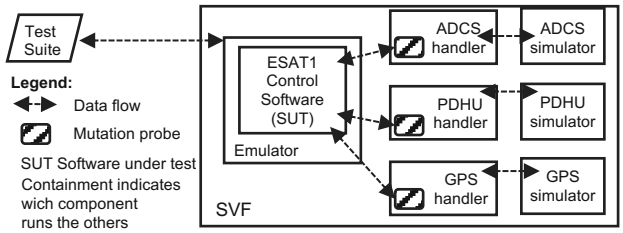

Fig. 5. Data mutation probes integrated into ESAIL.

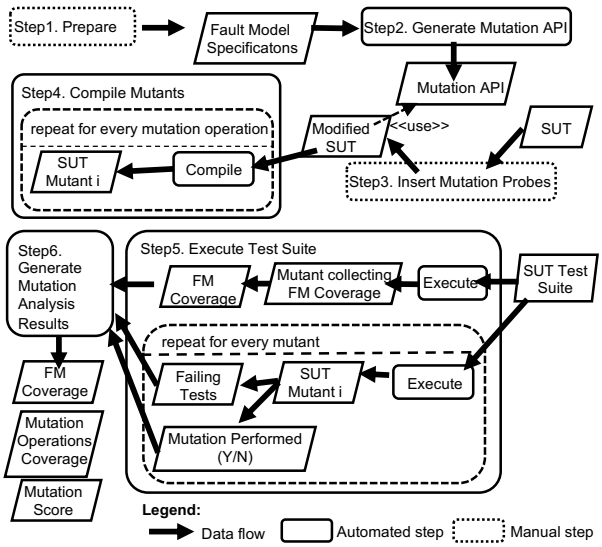

Fig. 6. The DAMAt process.

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.

4.0.1 Fault Model Structure. 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*; also, each CPS component may generate or receive different *message types*. For a single CPS, more than one fault model can be specified.

The DAMAt fault model enables the modelling of data that is exchanged through a specific data structure: the data buffer. 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*. Our current implementation supports six data representation types: int, long int, float, double, bin (i.e., data that should be treated in its binary form), hex (i.e., data that should be treated as hexadecimal). 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 .

Inspired by work on *abstract mutation analysis* [24], we have defined three metrics to evaluate test suites with data-driven mutation analysis: *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. Different from code-driven mutation analysis, these three metrics enable engineers to distinguish between possible test suite shortcomings, including untested message types, uncovered input partitions, poor oracle quality, and lack of test inputs.

Fault model coverage (FMC) is the percentage of fault models covered by the test suite. It provides information about the extent to which the message types actually exchanged by the SUT are exercised and verified by the test suites. Low fault model coverage may indicate that only a small portion of the integrated functionalities have been tested.

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).

5 DATA-DRIVEN MUTATION TESTING: DAMTE

The *test suite augmentation process* it consists of four activities *Identify Test Inputs*, *Generate Test Oracles*, *Execute the SUT*, *Fix the SUT*. It has the objective of increasing the score generated by the mutation analysis process.

FAQAS focussed on a methodology (i.e., data-driven mutation testing, *DAMTE*) that specifies how to rely on KLEE to generate test inputs that increase the fault model coverage and the mutation operation coverage. FAQAS does not address increasing the mutation score because infeasible in automated manner. We recall that two might be the reasons for a low MS: poor oracle quality and missing test input sequences. If the low mutation score is due to poor oracle quality, manual work is needed because automated approaches to automatically generate test oracles in the presence of system or integration test suites are not available. If the low mutation score is due to missing test input sequences (i.e., the software does not reach the state in which it could kill the mutant), manual work is required because existing test generation approaches (e.g., KLEE) might suffer from scalability problem that prevent bringing the system into a desired state; also, they cannot deal with systems whose components communicate through channels.

For the cases targeted by FAQAS (i.e., in the presence of fault model coverage and mutation operation coverage below 100%), test generation has the objective of generating test inputs that enable the application of all the mutation operators. We thus rely on an *extended data mutation probe* that invokes a version of the data mutation API that instead of mutating the data targeted by the mutation operator not covered by the fault model, includes a reachability assertion that is used to make KLEE find a test input that reaches the mutant code. The test input shall then be inspected by the engineer, who will need then to integrate it into his test suite.

6 FAQAS CASE STUDIES

The FAQAS toolset has been applied to six case study systems: ESAIL, *LIBGCSP*, *LIBParam*, *LIBUTIL*, MLFS, ASN1SCC. Table 3 provides additional details about each case study.

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. *ESAIL-CSW* is verified by unit test suites and system test suites that run in different facilities (e.g., Software Validation Facility [18], FlatSat [7], Protoflight Model [9]). We focus on the unit test suite and the SVF test suite because the other test suites require dedicated hardware. To address some of our research questions, all the mutants must be executed against the test suite, which is not feasible for the case of *ESAIL-CSW* due to its large size and test suite. For this reason, we have identified a subsystem of *ESAIL-CSW* (hereafter, ESAIL_S) that consists of a set of files, selected by LXS engineers, that are representative of the different functionalities in *ESAIL-CSW*: service/protocol layer functions, critical functions of the satellite implemented in high-level drivers, application layer functions.

Table 3. Overview of subject artefacts.

Subject	LOC	Test suite type	# Test cases	Statement coverage
ESAIL-CSW	74,155	System	384	90.38%
$ESAIL_S$	2,235	System	384	95.36%
LIBGCSP	9,836	Integration	89	63.10%
LIBParam	3,179	Integration	170	77.60%
LIBUTIL	10,576	Unit	201	83.20%
MLFS	5,402	Unit	4042	100.00%
ASN1CC	4,338	Unit	107	99.13%

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. MLFS is born from the need of having a mathematical library ready for qualification for flight software. 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 low resource environments such as space systems. Moreover, the compiler can produce a test harness that provides full statement coverage in the generated code, and therefore significantly improves its quality. In the context of FAQAS, we focus our analysis on the source code automatically generated by the ASN.1 compiler, rather than in ASN1SCC software itself.

Except for *ESAIL-CSW*, all subjects are compiled to generate executables for the development environment OS (Linux); we rely on the Gnu Compiler Collection (GCC) for Linux X86 [14] versions 5.3 and 6.3 for *MLFS* and ONE, respectively. *ESAIL-CSW* is compiled with the LEON/ERC32 RTEMS Cross Compilation System, which includes the GCC C/C++ compiler version 4.4.6 for RTEMS-4.8 (Sparc architecture) [1].

For the validation of each tool in the FAQAS toolset, we selected case studies presenting characteristics compatible with the requirements of the tool under test. Table 4 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. *Code-driven mutation analysis* (implemented by MASS) does not present any specific requirement except the availability of source code; for this reason, it has been validated with all the available case studies systems. *Code-driven mutation testing* (implemented by SEMuS), instead, requires the software under test to be comprised of components communicating through function calls (e.g., not network channels); for this reason, for SEMuS, we selected unit test suites. *Data-driven mutation analysis* (implemented by DAMAt) targets components communicating through channels, which are usually tested with integration and system test suites. For this reason we selected systems tested with such types of test suites. *Data-driven mutation testing* (implemented by DAMTE) aims to generate test cases that improve data-driven mutation analysis; however, since it is performed with symbolic execution tools, it presents the same limitations of SEMuS, that is, the analyzed part of the software under test should be comprised of components communicating through function calls. For this reason, it can target only libGCSP and libParam. We selected libParam because it is a higher-level library and thus more representative for this specific case.

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

Table 4. Case studies for the FAQAS activity.

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

7 EMPIRICAL EVALUATION

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

7.1 MASS

Table 5. Code-driven mutation analysis results: MASS mutation score.

Subject	**MASS Mutation Score (%)**
$ESAIL_S$ (System test suite)	65.95
$ESAIL_S$ (Unit+System test suite)	70.56
LIBGCSP	70.92
LIBParam	85.95
LIBUTIL	84.41
MLFS	93.49
ASN1SCC	80.77
Average	78.86

Table 5 provides the code-driven mutation analysis results. The mutation score computed by MASS for the case study subjects considered in our experiments was in line with the expectations of engineers. Both GSL and LXS have manually inspected a subset of the live mutants identified by MASS (18 for *LIBUTIL*, 7 for *LIBGCSP*, 9 for *LIBParam*, 19 for ESAIL). The inspection enabled industry partner to identify relevant shortcomings in their test suites:

- 30 live mutants were due to missing inputs (7 for *LIBUTIL*, 3 for *LIBGCSP*, 9 for *LIBParam*, 11 for ESAIL). In this cases engineers need to implement additional test cases that exercise the SUT with inputs not considered in the test suite. Of particular relevance are exceptional cases not being coverage (e.g., test suite was exercising the case of an error overflow caused by the parameter 'value' for a hash table but was not exercising the case of an error overflow caused by the parameter 'key').
- 12 live mutants were due to missing oracles (2 for *LIBUTIL*, 4 for *LIBGCSP*, 6 for ESAIL). In one case there was a missing oracle to verifies the correct encryption of all blocks encrypted by a certain function. In other cases the test suite was not verifying the output of the commands sent to the ADCS because verified when testing with hardware in the loop.
- one fault was detected.
- only two live mutants had ben reported (for ESAIL) and only eight mutants not relevant because concerning third party software (for *LIBUTIL*).

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

7.2 SEMuS

Table 6. Test suite augmentation results.

Subject	Live Mutants	Additionally Killed Mutants	Original MS (%)	Updated MS (%)
MLFS	3 891	697	81.80	85.06
ASN.1	2 219	1 729	58.31	90.79
ESAIL_S	1 041	NA	70.56	NA
Libutil	4 198	35	81.80	81.96

The empirical evaluation demonstrated the scalability of SEMuS for the case study subjects in which it can be successfully applied (i.e., ASN1CC, MLFS, and *LIBUTIL*). However, it is currently limited by the choice of compiling with LLVM only the source file under test (to limit the probability of compilation errors). Table 6 provides an overview of some of our results.

Our results also demonstrated the usefulness of SEMuS . Indeed, SEMuS enabled the identification of two fault in our case studies. Also, the generated test cases concerned inputs that are relevant (according to specifications) but not tested by the test suites.

7.3 DAMAt

Table 7. Data-driven mutation analysis results.

Subject	# FMs	FMC	#MOs-CFM	#CMOs	MOC	Killed	Live	MS
ESAIL-ADCS	10	90.00%	135	100	74.00%	45	55	45.00%
ESAIL-GPS	1	100.00%	23	22	95.65%	21	1	95.45%
ESAIL-PDHU	3	100.00%	29	24	82.76%	24	0	100.00%
LIBParam	6	100.00%	44	41	93.20%	37	4	90.24%
LIBGCSP	1	100.00%	33	21	63.64%	NA	NA	NA

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.

Table 7 shows the mutation analysis results of DAMAT. The empirical evaluation with ESAIL has demonstrated the effectiveness of the approach. Indeed, LXS has indicated that 57% out of the overall amount of 102 test suite problems detected by DAMAt were spotting major limitations of the test suite. Also, GSL has confirmed that the approach enabled the detection of relevant test suite shortcomings. One possible limitation of the approach is that it may introduce slow-downs that lead to non-deterministic failures when the test suite exercises brief interaction scenarios in which most of the operations performed concern the encapsulation of data into the network; this is what happened for the *LIBGCSP* case study subject.

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

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

7.4 DAMTE

DAMTE aims to address a task (i.e., test generation at system and integration level) that is particularly difficult to address with state-of-the-art technology (e.g., test generation toolsets based on symbolic execution). In particular, the test generation adopted in FAQAS (i.e., KLEE) requires manual intervention to specify which are the inputs to select thus preventing the automated generation of a large number of test cases. For code-driven mutation testing we have addressed this issue by relying on a template generator, which is difficult to implement for data-driven test generation because the identification of the function to test is hardly to automate (given that data-driven mutation analysis targets integration and system testing, it might be either the function with the mutation probe or another one). Also, and more importantly, it requires the compilation of the whole software under analysis through LLVM, which is often not feasible. For the reasons above, it is not feasible at the current stage to automate test generation of the whole software under test; consequently, it has not been possible to perform a large scale evaluation of the approach but only to focus on its feasibility analysis.

We relied on DAMTE to generate inputs for the *LIBParam* client API functions. The invocation of the *LIBParam* client API functions with the identified inputs enables the definition of integration test cases that exchange messages between the *LIBParam* client and the *LIBParam* server. The exchanged messages include data item instances that enable the execution of mutation operations that were not covered in the DAMAt empirical evaluation.

Overall, we conclude that the DAMTE approach may be feasible; however, it requires some manual effort for the configuration and execution of test cases which may limit its usefulness. The first step towards its large scale applicability is the improvement of underlying test generation tools and compiler procedures, such changes will facilitate DAMTE application to large projects without the need for manually creating test template files with dependencies.

7.5 Summary

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

Other limitations that need further research effort to simplify the adoption of the FAQAS toolset are the prioritization of mutants to be inspected, the need for a solution to compile whole SUTs with LLVM, the need for a solution to enable test generation in the presence of floating point

variables, the need for a working solution to enable test generation based on data-driven mutation analysis results.

8 CONCLUSION

The FAQAS activity had been motivated by the need for high-quality software in space systems; indeed, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Before FAQAS there was no work on identifying and assessing feasible and effective mutation analysis and testing approaches for space software. 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 to enhance mutation analysis scalability, identify mutants that are semantically equivalent to the original software or redundant, and automatically generate test cases. First, space software normally 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 prevents the use of static program analysis to detect equivalent mutants and automatically generate test cases.

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

The empirical evaluation conducted with the aid of industrial case study providers have highlighted the practical usefulness of the FAQAS toolset, which lead to the identification of relevant test suite shortcomings (test input partitions not exercised and missing test oracles) and bugs in the case study subjects.

REFERENCES

[1] Cobham Gaisler. 2020. RTEMS Cross Compilation System. https://www.gaisler.com/index.php/products/operating-systems/rtems.

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

[3] Oscar Eduardo Cornejo Olivares, Fabrizio Pastore, and Lionel Briand. 2021. Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3107680

[4] Marcio Eduardo Delamaro, Lin Deng, Vinicius Humberto Serapilha Durelli, Nan Li, and Jeff Offutt. 2014. Experimental evaluation of SDL and one-op mutation for C. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 203–212.

[5] Marcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. 2014. Designing deletion mutation operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 11–20.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[7] Jens Eickhoff. 2009. *Simulating spacecraft systems*. Springer Science & Business Media, Berlin, Germany.

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

[9] European Cooperation for Space Standardization. 2010. ECSS-E-HB-10-02A Rev.1 - Space Engineering, Verification guidelines. https://ecss.nl/hbstms/ecss-e-10-02a-verification-guidelines/

[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. 2008. ESA ISVV Guide, issue 2.0, 29/12/2008. ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/ecss/ESAISVVGuideIssue2.029dec2008.pdf

[12] European Space Agency. 2017. ExoMars 2016 - Schiaparelli Anomaly Inquiry. *DG-I/2017/546/TTN* (2017). http://exploration.esa.int/mars/59176-exomars-2016-schiaparelli-anomaly-inquiry/

[13] Free Software Foundation. 2020. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/
[14] FSF Free Software Foundation. 2020. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.
[15] Omid Givehchi, Klaus Landsdorf, Pieter Simoens, and Armando Walter Colombo. 2017. Interoperability for Industrial Cyber-Physical Systems: An Approach for Legacy Systems. *IEEE Transactions on Industrial Informatics* 13, 6 (2017), 3370–3378. https://doi.org/10.1109/TII.2017.2740434
[16] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227.
[17] Bernhard JM Grün, David Schuler, and Andreas Zeller. 2009. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 192–199.
[18] Y. Isasi, A Pinardell, A. Marquez, C. Molon-Noblot, A. Wagner, M. Gales, and M. Brada. 2019. The ESAIL Multipurpose Simulator. In *Onlin Proceedings of Simulation and EGSE for Space Programmes (SESP2019)*. https://atpi.eventsair.com/QuickEventWebsitePortal/sesp-2019/website/Agenda.
[19] Vaclav Jirkovsky, Marek Obitko, and Vladimir Marik. 2017. Understanding Data Heterogeneity in the Context of Cyber-Physical Systems Integration. *IEEE Transactions on Industrial Informatics* 13, 2 (2017), 660–667. https://doi.org/10.1109/TII.2016.2596101
[20] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2017), 308–333.
[21] 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
[22] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.
[23] NASA. 1998. Mars Climate Orbiter, Spacecraft lost. https://solarsystem.nasa.gov/missions/mars-climate-orbiter/in-depth/.
[24] Jeff Offutt, Paul Ammann, and Lisa Liu. 2006. Mutation testing implements grammar-based testing. *2nd Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), MUTATION'06* (2006), 12. https://doi.org/10.1109/MUTATION.2006.11
[25] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 354–365.
[26] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 69–80.
[27] David Schuler and Andreas Zeller. 2010. (Un-) covering equivalent mutants. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 45–54.
[28] David Schuler and Andreas Zeller. 2013. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374.
[29] D. Shin, S. Yoo, and D. Bae. 2018. A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 914–931. https://doi.org/10.1109/TSE.2017.2732347
[30] Thanassis Tsiodras. 2020. Cover me! https://www.thanassis.space/coverage.html
[31] 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.
[32] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 235–245.